UNIVERSITY OF OSLO
Department of Informatics

Lossless Data Compression Using Cellular Automata

Martin Haukeli

Network and System Administration

Oslo University College

May 23, 2012

Lossless Data Compression Using Cellular Automata

Martin Haukeli

Network and System Administration
Oslo University College

May 23, 2012

Abstract

This thesis presents an investigation into the idea of using Cellular Automata to compress digital data. The approach is based on the fact that many CA configurations have a previous configuration, but only one next generation. By going backwards and finding a smaller configuration we can store that configuration and how many steps to go forward, instead of the original.

In order to accomplish this an algorithm was developed that can backtrace a 2 dimensional CA configuration, listing its previous configurations. The algorithm is used to find a rule that often has previous configurations and also find an example where a matrix appears to become smaller by backtracing. The conclusion, however, is that when increasing the configuration size to trace the algorithm rapidly becomes too time consuming.

Contents

List of Figures

Chapter 1

Introduction

What if you could compress a 3GB movie down to 50MB? How much faster could it be downloaded? Can a compressed file be compressed further and further?

Todays Internet has grown very big and this is not only the number of nodes included, but also the data volume. In order to cope with the increasing amount of data everywhere in computer systems Data Compression is central. Data Compression helps users and system administrators lower the cost of storage and transfer by decreasing the volume required to represent the data.

A plentitude of algorithms have been developed to compress digital data. Most of which are specialized to a single type of data, such as text, image and sound. Note that several other categories exist for which special compression algorithms exist. Depending on the data type it may be allowed for some loss of the quality of the data. This allows the compression algorithm to make approximations to the original data(lossy), so that the Compression Ratio can be increased at the expense of the quality. Other algorithms however can be applied to any data type, these aglorithms are named General Data Compression. In such algorithms it is generally not acceptable with loss of precision (lossless).

Cellular Automata, a mathematical model of interacting cells in which time and states are discrete, has been proven to be able to simulate many different fields. For example in seismic simulation Cellular Automata has been used to predict the effect of earthquakes, achieving comparable results to previous algorithms in the field.[1]. Cellular Automata may be evolved with different rules and in different number of dimensions, allowing for a vast number of different Cellular Automata to be created. Cellular automata can be simulated in both bounded and infinite space. Cellular Automata can however only be simulated in one direction of time: Forward.

1.1 motivation

Creating and maintaining backups is an important System Administrator job. The backups may be big and it may be required to keep them for an indefi-

nite amount of time. Because of this it is essential to compress backup data. This however could be done off-line, as nobody will usually want to see the backups unless some rare event has occurred.

Current compression algorithms can only compress general data somewhat, and cannot compress data further. What if it was possible to spend more time compressing data to achieve greater compression? This would not only affect Backups, files that are to be distributed to a large amount of peers could also gain benefit from more compression. It is therefore important to investigate new methods and ideas for compression.

Cellular Automata appears to be able to affect matrices in what seems unpredictable and chaotic ways. Yet, by altering the rules the Cellular Automaton can be tailored to change the data in varying ways. An interesting category of rules named Fredkin's Replicators contain rules that will always result in the original pattern being copied an infinite number of times (Replicated)[2].

problem statement

This thesis will try to create a Cellular Automaton based compression algorithm for lossless general file compression with the hopes of being able to trade CPU time for increased compression ratios.

1.2 objectives and methodology (high level overview)

- Find a method to develop CA algorithms that can compress data.
- Review previous CA models to see if any of them can easily be used for lossless data compression.
- Try to develop a proof-of-concept CA compression algorithm.

Chapter 2

Background

2.1 Data Compression

General Data compression is a way of encoding data so as to eliminate redundancy and achieve smaller data sizes while still retaining the original data or the original meaning of the data. Redundancy can be seen as the amount of entropy in the data, more entropy equals less redundancy.

Lossless data compression, as opposed to lossy, guarantees that the original data can be restored from the compressed data. In lossy compression it may be that only an approximation of the original data can be restored from the compressed data. When storing data such as images, video or audio the use of lossy compression makes sense as the original meaning could be retained even if the actual bits are not the original. For other data types such as applications, lossy compression cannot be applied as it would change how the program operates.

In order to do this, we can search for repetitive parts of data, and replace them with short-hand representations. Random data cannot be compressed, however. [3]

The huffman coding works by assigning each symbol a new representation form. The most common symbols are given the shortest representation form. This can most easily be seen with a binary tree.[4]

For example the ASCII text "HELLO WORLD, THIS IS THE COMPUTER TALKING" gives us the probabilities:

H 3/41	E 3/41	L 4/41	O 3/41	' ' 6/41
W 1/41	R 2/41	D 1/41	, 3/41	T 4/41
I 3/41	S 2/41	C 1/41	M 1/41	P 1/41
U 1/41	K 1/41	N 1/41	G 1741	A 1/41

Using these probabilities we can create a huffman tree. Note that several possible huffman trees exists. Figure 2.1 is one such tree.

Huffman coding is an example of statistical compression. It works well on data where we have a statistical model, like text from the English language.

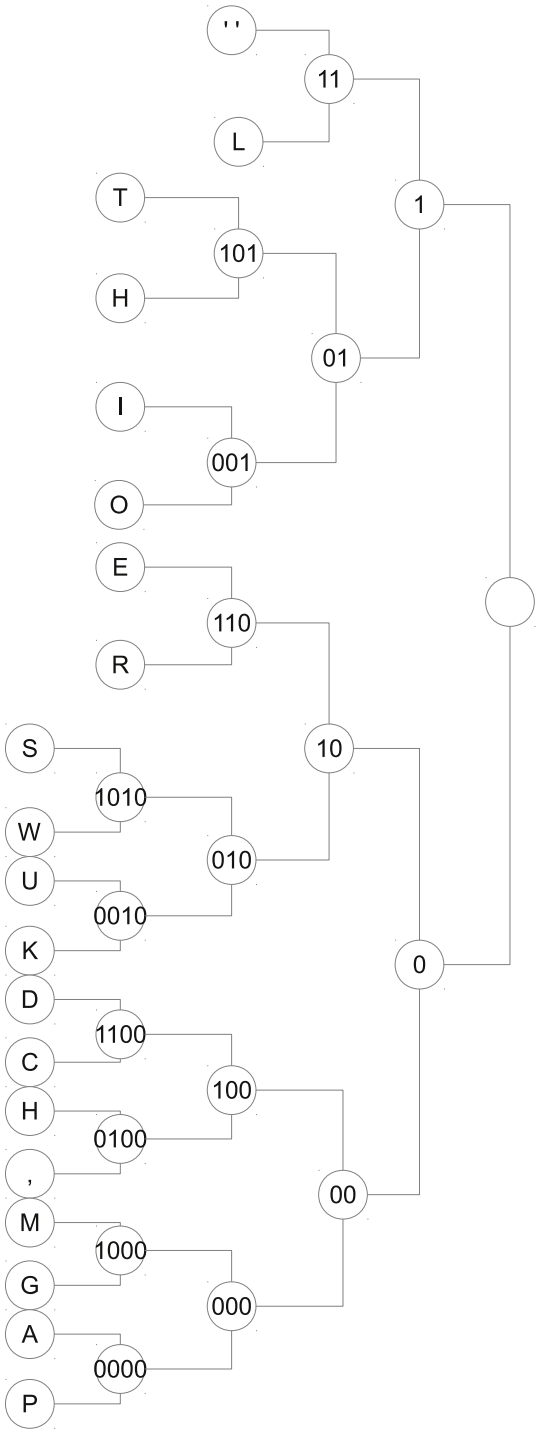

Figure 2.1: A huffman tree for the text "HELLO WORLD, THIS IS THE COMPUTER TALKING".

For other cases such as compressing never before seen data we have other methods.

The dictionary method's main idea is to represent each string of symbols as

a token found in a dictionary. This dictionary can either be dynamic, allowing additions, or static.[5]

Information Theory

The beginnings of information theory was laid out by Claude Shannon in 1948 and describes information in a quantitative way.

The entropy of some data 'd' is how much information this data contains. We can say that a data source with more entropy is more random. This way we can use the notion of entropy to describe how well a block of data is compressed.[6]

General data compression schemes

For most popular data formats exists specific compression tools, such as JPEG for pictures, lame for music and MPEG for movies.

For general data, when the content cannot be recognized, such algorithms prove useless and one must instead look at how the bit patterns look like. Most general data compression schemes are based on dictionaries; Finding what bit-strings appear the most and replacing those with shorter versions, thereby removing redundancy. When all redundancy is removed the file is said to be optimally compressed and contains only random data (except the header for decompressing).[7]

One of these methods, RLE run length encoding, achieves compression by replacing a string such as "aaabbbcccaaaa" with "3a3b3c4a" which is shorter. Of course if the string to be compressed is more random, say "abjgaski" then this method will not be able to save anything.

On Linux with KDE, the available compression schemes in the default compression tool "ark" were: zip, 7-zip, rar, gzip, bzip, tar-z, xz and lzma. Comparing methods of lossless data compression has been done at several occasions. Generally the algorithms perform at a comparable level.[8][7]

2.2 Cellular Automata

Created in 1940's by John Von Neumann, cellular automata is a mathematical model to simulate evolving cells in a lattice of infinite space where time is discrete and all rules are only applied locally at cell level. The original purpose of Cellular Automata was to study self-replication.[9][10] Cellular automata can be simulated in any dimension (1, 2, 3, etc.). But, in this thesis we will only consider 1 and 2 dimensional space.

The evolution of cellular automata springs out of an initial configuration, also known as seed, that along with its rules determines the rest of its history. Knowing the seed and rules for evolution, any following generation can be calculated.

The rules of a cellular automata describe how cells are to be evolved. The state of a cell at time t is usually dependent on its neighboring cells at time t, and its state at time t-1.[11][12] Which cells belong to its neighborhood is part of the rules, but is limited by its dimensionality.

The cells of a CA each have one of a finite number of states. The most common CAs have 2 states (binary CA) where 0 represents a dead cell and 1 represents a live cell. In this thesis we will focus on binary CA.

In a 1 dimensional cellular automata the field is usually represented as an array.

```
[X0XX00X000000X00X]
```

Here the X's represent live cells, O's are dead cells. The neighborhood of 1 dimensional cellular automata can only be to the left and/or right, but any number of cells specified by the rules. Toroidal 1 dimensional CAs are not uncommon.

Two dimensional cellular automata can be represented by a matrix such as Figure 2.2, the white squares are live while the grey are dead.

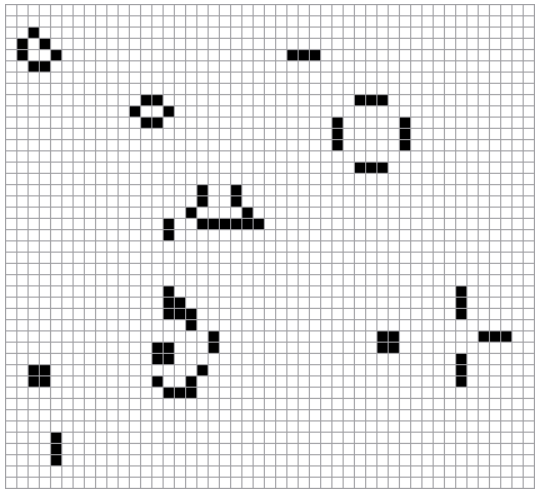

Figure 2.2: 2d Cellular Automata (Game of Life, B3/S23).

Here the neighborhood can be either along the square axis and/or along the diagonals. Note that neighborhoods do not need to be symmetric.

In this thesis only Cellular Automata in finite space will be studied and in the form of a Null Boundary; Cells outside the boundary will be considered permanently dead.

The rules for binary CA is generally written as Bx/Sy (Golly format[13]), where x is the number of neighbors required for a dead cell to become alive(Born), and y is the number of neighbors a live cell needs to stay alive(Survive). Both x and y can contain several numbers and even the same numbers or be empty. Figure 2.3 shows several valid binary CA rules. Note that rules for CA with more than 2 states are written differently and are not considered in this thesis.

The x and y part of the rule may contain numbers from 0 to 8, that is up to 9 numbers in each part. Using this information we can calculate the amount of different binary CA rules. There are thus $2^{2*9} = 262144$ different rules for binary CA in the Moore neighborhood.

`B1/S123` A rule where dead cells become alive with 1 neighbour and survive with 1,2 or 3 neighbors. Will grow to infinity at the "Speed of Light".

`B/S3` A rule where dead cells cannot become alive, but live cells with 3 neighbors stay alive. Note that with no births the CA can never grow.

`B3/S012345678` A rule named "Life without Death" where cells never die and new cells are born if they have 4 neighbors.

`B2/S` A rule named "seeds" where cells die every generation. Due to the low requirement for birth it will usually grow extremely fast.

Figure 2.3: Example binary CA rules in the Moore neighborhood

2.2.1 Moore neighborhood

This paper will only consider the Moore neighbor hood for 2d Cellular Automata. In this neighborhood model all cells have 8 neighbors as can be seen in **??**.[10] Note that in finite space cells that are in the corners or along the border have 3 and 5 neighbors respectively.

2.2.2 Speed of Light

In Cellular Automata movement speed is measured by how many cells the pattern moves per generation. The maximum speed is 1 cell per generation and the Speed of Light in Cellular Automata has thus been defined as 1 cell/generation.[10]

2.2.3 The Game of Life

One of most popular rules for cellular automata is John Horton Conway's Game of Life(1970). It is a 2 dimensional cellular automata with the rule

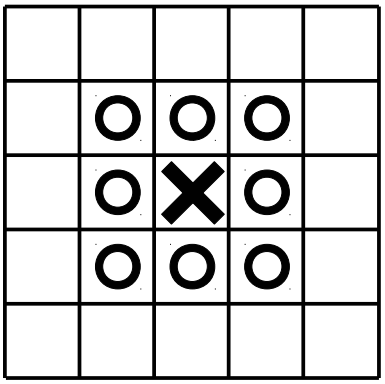

Figure 2.4: The moore neighbourhood for a cell X, this cells neighbourhood is marked with o's

B3/S23. This rule was chosen by Conway because it exhibited interesting features when simulating.[12]

Simulation:[10]

- If a live cell has less than 2 neighbors, it dies of loneliness.
- If a dead cell has exactly 3 neighbors it is born (becomes live).
- If a live cell has more than 3 neighbors, it dies of overcrowding.
- In all other cases, no change is made.

There has been alot of research into different aspects of the Game of Life. It has been proven Turing Complete(allowing Universal Computability) and self replicating patterns have been found, as well as frequent patterns and a huge collection of interesting patterns.[14][15]

There are many patterns in The Game of Life that cannot appear naturally because they contain some cell configuration (Orphan) that the rule B3/S23 does not generate. These patterns are named Gardens of Eden, they have no previous pattern, but as all other have a next pattern.[16]

As of writing, the currently known smallest Orphan was found by Marijn Heule, Christiaan Hartman, Kees Kwekkeboom and Alain Noels. This Orphan fits a 10x10 area.[17] There are

$2^{10^2} = 1267650600228229401496703205376$

different 10x10 areas in Game of Life.

By reading a 1MB file (1MB = $1024 * 1024 * 8bits = 8388608$ cells) into a square area (size $\sqrt{1024 * 1024 * 8} \approx 2897$), there is a chance that it will contain

at least one of this Orphans. If all the possible 10x10 areas where equally likely, then the chance of a single 10x10 area being the given Orphan would be

$$\frac{1}{1267650600228229401496703205376}.$$

A square matrix of size 2897 can hold $((2897-10)+1)^2 = 8340544$ 10x10 areas. Thus we can calculate the chance of a 2897 matrix containing a given 10x10 matrix as

$$\frac{8340544}{1267650600228229401496703205376}.$$

If we also search for all the other known Orphans, this chance will increase futher. It is therefore given that many files are Gardens of Eden (when using Game of Life).

2.2.4 Evolving Cellular Automata

Because time in Cellular Automata is discrete it is possible to iterate over time in steps of one generation at a time. To go from one time step to the next, the cellular automata rules must be applied once to every cell in the configuration. This can be seen as a function to evolve the cellular automata. Figure Figure 2.5 shows how a configuration is evolved in Game of Life, it is important to note that the evolution function can only move time forward and that there is always exactly one next generation.

Game of Life is defined with the rule B3/S23, this means that a dead cell becomes live when it has 3 live cells as neighbors and a live cell survives if it has 2 or 3 live neighbours. Not that this means that a live cell dies if it has 0,1,4,5,6,7 or 8 live neighbours. In Figure 2.5 the cells that will die are first marked with red, the 2 light red dies because it has 1 neighbour and the dark red (center cell) dies because it has 4 neighbours. Next the cells that are born is marked with green, 2 dead cells have 3 live neighbours each.

When all cells have been checked the actual cells are changed to give the next generation $\tau + 1$. Note that a cell can only be changed once each time step as all cells are updated at the same time.

Even though there is only one next generation for each configuration, several previous generations may exist.

2.3 The basic idea of compression using CA

In order to compress data, the idea is to lower the number of bits required to store it. That is representing information as effectively as possible. Compression is achieved by eliminating redundancy, thereby increasing the entropy of the data.[5]

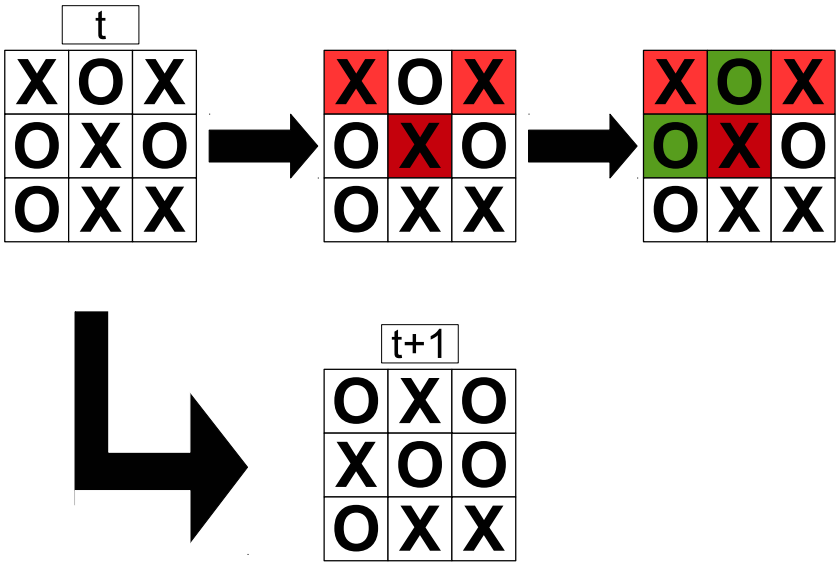

Figure 2.5: How to evolve one configuration to the next generation in Game of Life (B3/S23)

Since any data can be generated from a function, although complex, it should be possible for any data to be represented as that function instead. Investigation into the generative properties of cellular automata has shown that Cellular Automata can be used to generate random data with high efficiency. [18]

It has further been shown that Cellular Automata can be used as a universal finite pattern generator. Meaning that any given finite pattern may be generated by a Cellular Automaton. It is however not given if this can be applied to binary CA or 2 dimensional CA space.[19]

The Idea
Since Cellular Automata can generate random data sequences and one can only evolve a CA forward it means *one could in theory represent some data with a configuration that leads to the original data*. An algorithm to recover the original data would be needed, and it would need to know which rule to use and how many generations to evolve. Now, given that one could find a configuration in some CA rule that leads to the data, if it is smaller than the original then the data has in fact been compressed.

Thus, the problem is to find a configuration that requires less bits to represent and evolves into the data to compress, possibly after many generations. If such a configuration is found then it can be stored, along with number of generations to go, instead of the original data. It has been shown that 2D Cellular Automata express interesting matrix properties, especially in the group of CA that are non-uniform.[20]

One important aspect of cellular automata in regards to matrices is the fact that no matter the size of the matrix, all cells are checked and updated at every step. This means that in only 1 or 2 steps a matrix, independent of the size, will change a lot. Of course this depends on the configuration and rule used.

In data compression, entropy is seen as the limiting factor. If the entropy is too high, the data cannot be compressed further. But in cellular automata compression, based on this idea, one can say that the ratio of Gardens of Eden is the limiting factor as it limits the movement backwards.

2.4 Cellular Automata Transforms

Cellular Automata Transforms (CAT) is a method for finding cellular automata that can be used to transform data into some other more desirable form. It has been able to both compress multimedia data and encrypt text.[21]

The way Cellular Automata Transforms has been used for compression is lossy. The original data was transformed by the CAT to a compressed form that is only an approximation of the original data, because of this the method cannot be directly applied to general data which requires lossless compression.

Because of this fact, and time constraints, Cellular Automata Transforms were not investigated further in this thesis.

2.5 Reversible Cellular Automata

A cellular automata is said to be reversible if given any configuration there is exactly one previous configuration. Finding this previous configuration follows the same procedure as finding the next.[11]

Trying to find the previous configuration of a non-reversible CA however is not as trivial, previous attempts usually includes brute force as the only way to find any of the previous configurations if they exist. It is also unclear if a previous configuration exists.

Since a previous configuration always exists for reversible cellular automata, it seems likely that they may be used to solve my generative problem.

2.6 Java

Java is an Object Oriented programming language with syntax much like c++. Java code is compiled into byte code which is read by a virtual machine (a JVM, java virtual machine). A compiled java file becomes a '.class' (or '.jar' for several files) which the JVM can read. Since code is not compiled to machine code any computer with a compatible JVM can run java code. Java is as such platform independent and easy to distribute.

In its early days, java was quite a bit slower than code compiled with c. In recent years this has changed a lot however and java has become almost as fast (and sometimes the same) as C.

2.7 Genetic Algorithm

The genetic algorithm(GA) is part of evolutionary algorithms, a field of algorithms inspired by nature and evolution. GA is a search and optimization algorithm that combines the strengths of brute force and local optimization. It is a good overall approach to many problems, but is usually beaten when specific algorithms written for the task exists.[22]

The algorithm simulates evolution of *candidate solutions* as genes fighting for survival. The more fit genes are allowed to create offspring, while the bad genes are weeded out. This process will take the fit genes and combine them, while discarding the bad genes. In this way the genetic algorithm will converge to a single solution. This solution is usually a good solution, but finding the absolute best cannot be guaranteed.

In order to simulate a real evolution the algorithm requires several operations to be defined:

- A crossover function to breed 2 (or more) genes to create offspring
- A mutation function to affect the offspring randomly
- A fitness function that rates the gene based on how good it is (in our case how close it is to the solution)
- A selection to decide which genes to breed, usually based on their fitness.

Although the Genetic Algorithm is quick to find locally good solution, the global optimum may never be found. It is also so that the Genetic Algorithm will investigate the same gene several times if it comes up.[23]

2.8 Previous algorithms for Reversing Cellular Automata

As of writing there are 4 known algorithms for "listing preimages", that is finding the previous configurations (Ct-1) that lead to a specified configuration(Ct) in rule R. All of these where designed for 1 dimensional CA and can be solved in linear time depending on configuration size. All of them also make use of a model for permuting the total number of combinations of a diagram known as a De Brujin Diagram.[24]

In this thesis we will consider only the two latest of these algorithms "Trace and backtrack" and "Count and List" as they are newer and contain the major

parts of the other two.[24]

The first 2 steps of these algorithms is to create "preimage diagrams" for each cell and then link these diagrams together to create what they call a "preimage network". After traversing the network we find all the previous configurations as successful paths. Figure 2.6 shows an example Preimage Network with one valid path.

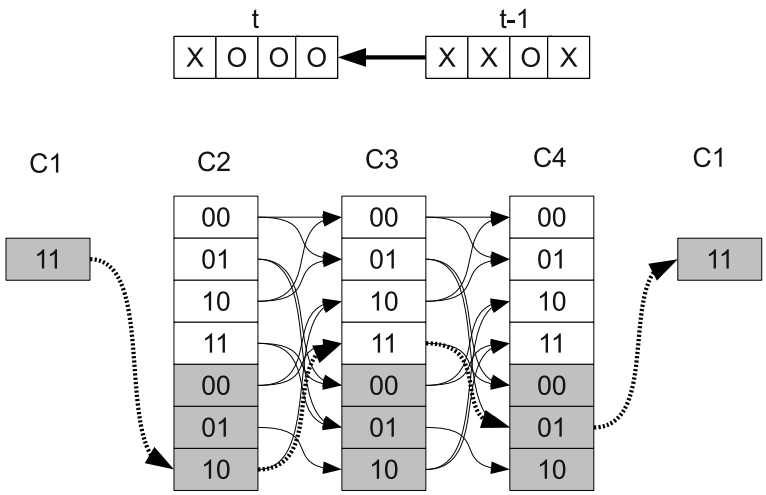

Figure 2.6: An example trace in a Preimage Network for 4 cells in a cyclic array. Grey boxes indicate a live cell value, white is dead. The only valid trace is given by the thick dashed line, as it is the only line reaching from the first cell to the last cell. Note that because this is a cyclic array, the first cell is also the last.

Because these algorithms use 1 dimensional configurations it is possibly to trace simply from left to right (or right to left) as each cell can only have neighbours to the left or to the right that affect each other. If one were to consider 2 dimensional CA space one would have to iterate in all directions of the neighborhood. In the Moore neighborhood, where each cell has 8 neighbors, this would mean 4 directions of tracing.

The complexity can be described as the number of links between cells needed to be inspected to know if a solution is valid. When considering finite one dimensional CA the amount of links equal the size and the complexity is linear. In a square finite 2 dimensional CA with the Moore neighborhood and size **s** the number of links is given by

$$L_s = \sum_{i=2}^{s}(8i - 10), (s > 1, \text{for } s = 1 => L_s = 0).$$

Because of the abundant growth of links to inspect, and time constraints, the algorithms were not studied further in this thesis.

Chapter 3

Design

In this chapter the implementation of each algorithm is specified. Most of the actual code can be found in the appendix. Each algorithm tested has it's own section in this chapter where design and implementation details are specified.

Test System

Operating System: 32bit Ubuntu 11.10 with PAE kernel

CPU: Intel Core i7 3820 (4 cores @ 3.6 GHz)

Memory: 16GB

Java version: OpenJDK 1.7 (1.7.0_147-icedtea)

3.1 Reversible Cellular Automata

In order to test the possibility of using Reversible Cellular Automata for compression an implementation was attempted in Java. The implementation was developed for both second order and block cellular automata. The algorithm will iterate over time step by step and attempt compressing the current configuration at every step.

3.1.1 Block Cellular Automata

For block cellular automata, the critters rule and the tron rule was used.

The implementation of the function to calculate the next generation, by applying the local rule to each block, can be found in appendix subsection 7.6.2.

Critters rule

For each block of 4 cells an operation is carried out depending on the number of live cells in the block.

- If there are exactly 2 live cells, no operation is carried out.
- If there are 3 cells, the block is flipped (rotated 180 degrees).
- Unless the block had 2 cells it is inverted (all live cells become dead and dead cells become live).

The implementation of the Critters local rule can be found in appendix subsection 7.6.4.

Tron Rule

For each block of 4 cells, if all the cells in the block is the same state then the block is inverted.

The implementation of the Tron local rule can be found in appendix subsection 7.6.3.

3.1.2 Second-Order Cellular Automata

Because second order cellular automata requires 2 consecutive states to move in any direction, a randomly generated matrix was used to represent $\tau - 1$ while the input data to be compressed was set as τ . The algorithm then iterates over time to get $\tau + 1, \tau + 2$ and so forth. As with block cellular automata, each configuration encountered is compressed.

Different rules were tested, among them 'B234/S234' because it makes the configurations look very different at each step. The implementation of the function to evolve the CA, $\tau + 1 = \tau - 1 \oplus \tau'$, can be found in appendix subsection 7.6.1.

3.2 Genetic Algorithm

The genetic algorithm was implemented in Java to search for previous configurations of an input configuration and given rule. Two crossover and two mutation methods where implemented. Tournament selection was used to select which genes from the population to breed.

3.2.1 Crossover

The two crossover operations implemented are named XOR and random. The algorithm randomly chose which operation to use, with 50% chance for each operation.

XOR crossover

In XOR crossover the XOR operation is used on the 2 parents to give 1 offspring. The XOR ($\oplus$) operation is the exclusive 'OR' of the input. An example of how the operation works is shown in Figure 3.1.

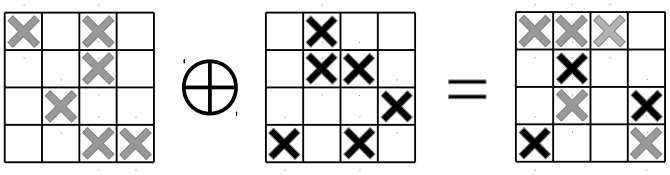

Figure 3.1: XOR crossover with 2 parents producing 1 offspring

Random crossover

Random crossover was implemented as iterating over both parents choosing randomly either the first or second parent to copy from. At each step a new random number is drawn giving 50% chance to both. Hence the offspring will on average consist of 50% of each parent. Figure 3.2 shows how two parents may be combined by random crossover.

3.2.2 Mutation

After each crossover mutation is applied in 50% of the cases. The operations implemented is Flip Mutate and Edge Flip Mutate.

Flip Mutate

Flip mutate is a simple operation where each cell in the configuration is inverted by a configurable chance. On average the percentage of cells inverted is equal to the chance set. Although the operation is simple it will successfully randomize the configuration a configurable amount. This is good because we can then increase the mutation strength if we need more exploration.

Edge Flip Mutate

Edge flip mutate was derived from flip mutate to only iterate over those cells that have at least 1 live neighbor. Each of these cells are inverted by a configurable chance. Figure 3.3 shows which cells have a chance to be inverted in a given configuration. This mutation boosts exploration less than Flip Mutate as the randomization is only applied to some of the cells. It does however mimic the evolution of CA a little in that only the edges can be moved, new cells will not appear away from the other cells.

3.2.3 Tournament Selection

Tournament selection is a selection operation that favors the more fit genes. It operates by selecting a set amount of genes (contestants), compares them and selects the best. In order to get more than 1 gene the operation is run again from the start independent of the previous run.

A good suggestion for the number of combatants is 3 as this keeps the algorithm from converging too fast or too slow. [22]

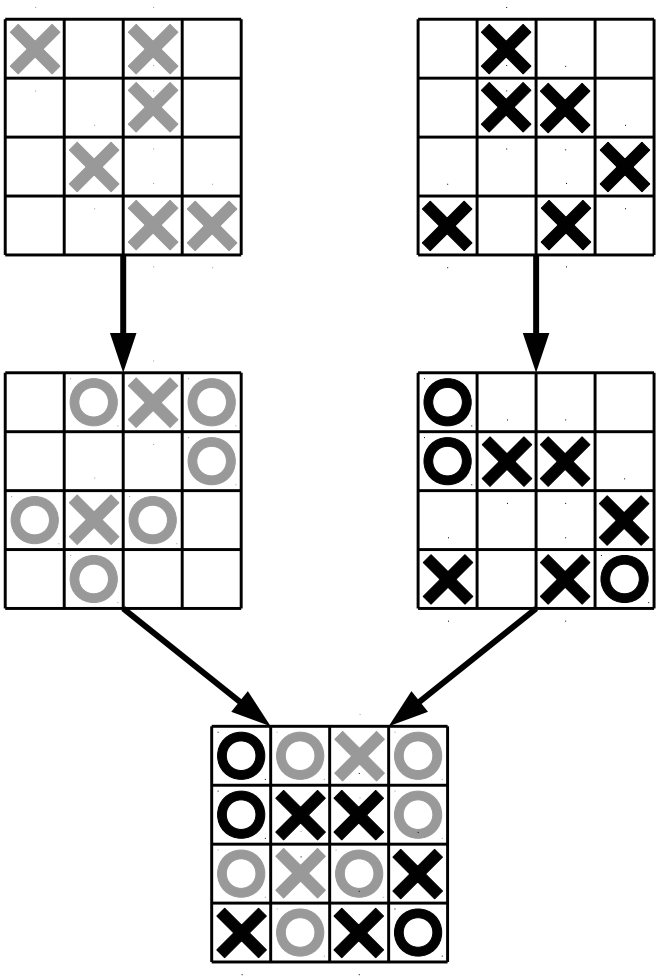

Figure 3.2: Random crossover with 2 parents producing 1 offspring

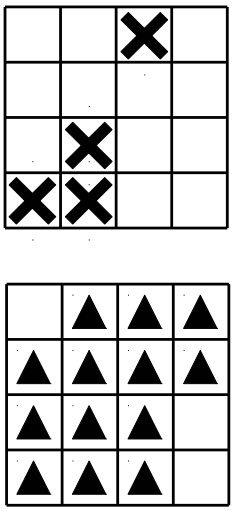

Figure 3.3: From the configuration above, the cells with a triangle have a chance to be inverted by edge flip mutate

3.3 Colouring Algorithm

In order to find a new way to search for previous configurations, a test was carried out to find how one could accomplish this with only pen and paper. A deduction was made that the person solving it would look at each individual live cell and see how it can have appeared. To mark the possibilities for each cell, colour pencils were used. Figure 3.4 shows how one configuration was colored.

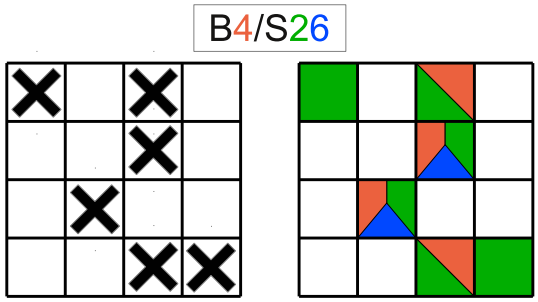

Figure 3.4: Coloring a 4x4 matrix

Then, by guessing permutations of the colors, each live cell was given neighbours to satisfy its requirements. If a requirement could not be met a new permutation was guessed and the old was discarded. Although a slow process requiring several guesses, the algorithm did work out.

In order to distinguish the cells that have been added from the original cells a convention for representation was devised. Table Figure 3.5 shows what each symbol means. It is important to note that a finally dead cell is not the same as a cell that dies now, the former is dead in the current generation and stays dead in the next, while the latter is alive in the current but dead in the next. New live cells that are added to satisfy another cell must be added as 'dies now', because the algorithm wants to get to the next generation it was given.

First a permutation of colors must be guessed, here the number of permutations depends on the number of live cells, the length of the rule and the cell positions. Note how in Figure 3.4 the corner cells can only be green because there is only 3 available neighbours for that cell, hence the only rule that fits is 2 which is green. Using the colormap permutations of colors can be guessed. To represent a guess the color chosen for each cell is listed in order, counting cells from the top left to bottom right. Figure 3.6 shows the order of counting cells, note that in this case the dead cells are skipped.

Figure 3.7 shows another configuration with a corresponding colormap. A sample guess could thus be OOXOXO, which would look like Figure 3.8. Here the symbols from Figure 3.5 is used together with the color. The next step is to try satisfying all the cells at the same time.

Symbol	Meaning
X	Finally alive; Alive in both the current and the next generation
O	Undecided; Unknown in the current, dead in the next
⧖	Dies now; Alive in the current generation, dead in the next
⊖	Born; Dead in the current generation, alive in the next
⧈	Finally dead; Dead in both the current and the next generation

Figure 3.5: Symbols devised for backtracking

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 3.6: How counting the cells was done

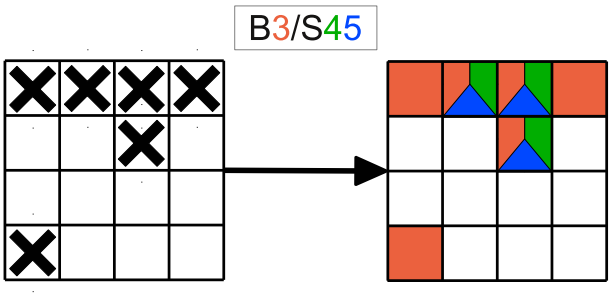

Figure 3.7: Another configuration and corresponding colormap

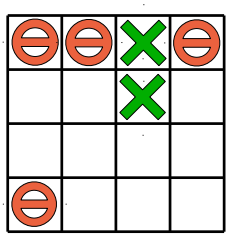

Figure 3.8: One permutation of a colormap ready to be tested

Figure 3.9 shows several guesses to satisfy a colormap, the rightmost holds a valid solution. Note how it is possible to discard an approach when a color cannot be satisfied, but before all cells have been tested. It was also found that several guess could have been eliminated when looking at multiple neighbouring colors at the same time. For example in Figure 3.9 if the corner is orange then none of its neighbours may be orange, otherwise it cannot get enough neighbours.

The pen and paper approach was coded in Java and tentatively named the Colouring Algorithm, from the use of colour pencils. The algorithm was implemented in Java with an object oriented approach. The coloring part of the algorithm was implemented in a class named ColorMap while the algorithm for attempting to solve each permutation was implemented in a class named PermMap.

The code was written with thread support for multi-core systems. The threading is based on a monitor and worker model, where one thread creates worker threads that do the actual jobs and monitor their states. A separate thread works on pushing work onto an agenda, which the monitor reads and distributes to the workers. The system also does limited load-balancing by increasing the number of workers when there is a lot of work and decreasing the amount when there's little load per worker. Appendix **??** has the code used in conjunction with Perm Map and Color Map.

3.3.1 Color Map

In order to limit the number of iterations of the algorithm, It was decided to iterate only over the permutations of how each cell can have been formed using the colouring approach described above. For this the "Color Map" was created to represent such colored matrices as shown in Figure 3.4. The color map describes all possible ways that could have caused each live cell being alive, either by birth or survival. Using this information iteration is possible over the permutation of colors.

For example, a brute force approach to iterating over all 6x6 binary matrixes would be 2^{6^2} iterations. If 12 of the cells are live, and there are 3 colors,

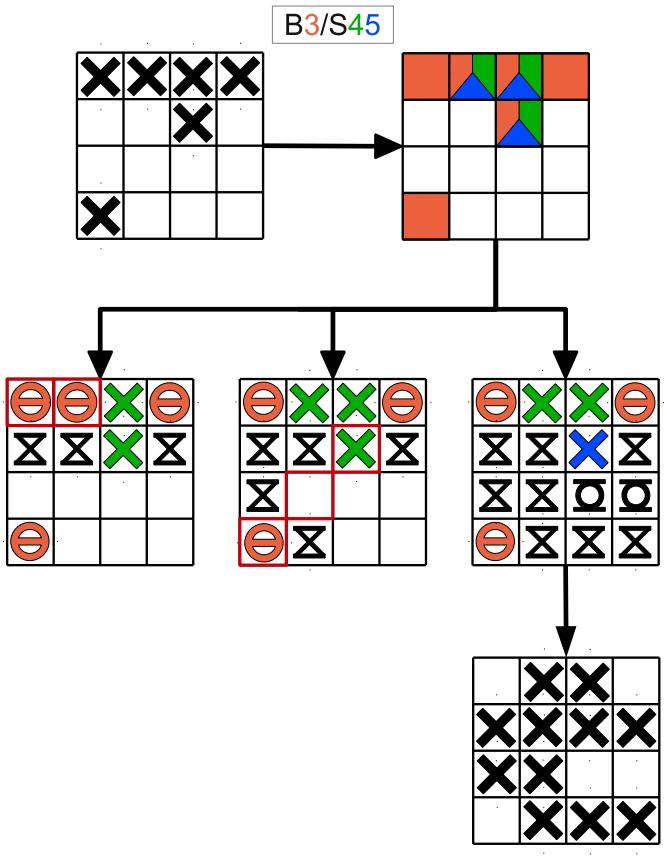

Figure 3.9: Several attempts at solving colormaps, one attempt has a solution. The red squares indicate colors that don't fit together, resulting in the permutation beeing discarded.

then the colormap would have 12^3 permutations (roughly $2^{3.3^2}$ for comparison).

Note that a permutation of the color map is not a solution, but rather a possible explanation of how the cells can have entered their given state. In order to find a solution the permutation must be satisfied as in Figure 3.9

In order to further eliminate iterations it was also looked at cells that exist in corners or next to borders. For these a check was made that each color given to them is possible given their maximum amount of neighbours. For example the corner cells can only have 3 neighbours, while cells at the borders can only have 5.

Another approach used to eliminate iterations was to look at cells that depend on each other. If one cell requires an amount of neighbours and of these cannot be in the required state, then that color can be eliminated.

The implementation of Color Map is based on an array containing the possible colors for each cell. Each color is a Byte that references one of the numbers in the rule used. The rule B3/S23 has 3 numbers = 3 color rule. Here 0 = B3, 1 = S2, and 2 = S3.

Listing 3.1: Iteration over the ColorMap is provided by the **gotoNext()** method

```
1   public void gotoNext(){
        if(numColors <= 1)
3           return; // Nothing to do.

5       if(!hasNext())
            return;
7
        add(colors.length-1);
9       //this.gotoRandom(); an option to iterating..

11      for(Elimination e : eliminations){
            if(e.isEliminated()){
13              do {
                  add(colors.length-1);
15              } while(e.isEliminated());

17          }
        }
19  }
```

The **gotoNext()** method calls the **add()** method causing an increment in the current colors. It then checks all the eliminations that applies to this Color Map. If any of them trigger then **add()** is called again. This process repeats untill the elimination that triggered is satisfied.

Listing 3.2: The method **add()** is a recursive method that increments the color of the cell in the last position

```
1   private void add(int pos){
        if(pos < 0 || pos >= colors.length){
3           this.broken = true;
```

```
            return;
        }

        if(aviColorsCounter[pos] < aviColors[pos].size()-1){
            colors[pos] = aviColors[pos].get(++aviColorsCounter[pos]);
            return;
        }
        else {
            aviColorsCounter[pos] = 0;
            if(aviColors[pos].size() == 0)
                return;
            colors[pos] = aviColors[pos].getFirst().byteValue();
            add(pos-1);
        }
    }
```

The **add()** method is a recursive permutation method that iterates the possible colors of each cell. By increasing the color, from right to left the method acts like a simple adder. Note however that since each cell may have different colors available the method needs to check which colors each cell may have. This information is stored in the array avi Colors.

In order to represent eliminated colors, a new class was created to support adding cells and which color was eliminated. This way an elimination can be made which includes several cells and colors. The implementation of the Elimination class can be found in appendix section 7.2 on line 105. The method **optimize()** that parses the Color Map and makes eliminations can be found in the same appendix on line 234.

Because the **optimize()** method may create duplicate eliminations another method was deviced to weed out the eliminations. The method **optimizeEliminations()**, in appendix section 7.2 on line 54, compares all the eliminations registered to see if any of them are equal or contain eachother.

On complexity

When making the colormap we can calculate how many different permutations of colors exist.

This is given by a simple combinatorial formula:

$count = colors_0 * colors_1 * colors_2 * ... * colors_n$, $colors_n$ being the available colors of cell **n**, etc.

To calculate the number of permutations for a 3x3 matrix with 4 cells as shown in Figure 3.10:

1. $cell_0$ can have 3 different colors
2. $cell_1$ can have 4 different colors
3. $cell_2$ can have 1 color

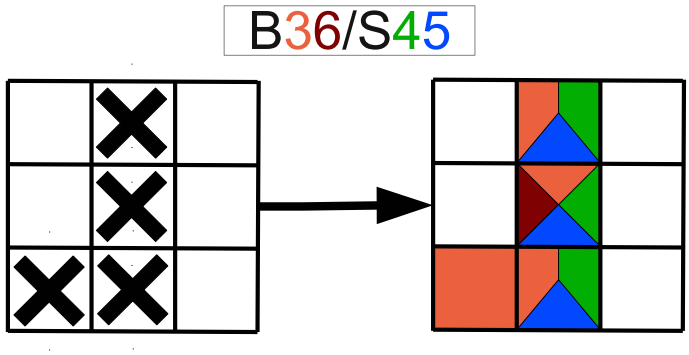

Figure 3.10: Color map for a 3x3 matrix with 4 cells

4. $cell_3$ can have 3 different colors

The calculation becomes

$count = colors_0 * colors_1 * colors_2 * colors_3 = 3 * 4 * 1 * 3 = 36$ different ways to distribute the colors.

This number however is then affected by which eliminations that have been made. If for example $cell_0$ cannot be blue while $cell_1$ is orange, then all the permutations that contain $cell_0$=blue, $cell_1$=orange must be subtracted. In this case there are $1 * 1 * 1 * 3 = 3$ such permutations, which gives the new total permutations as 33.

If however it was found that in addition $cell_2$ cannot be orange if $cell_1$ is orange, then a new subtraction must be made. This time however it becomes much more difficult, as the overlap from the first elimination must not be subtracted again. This can be visualized by a venn diagram as seen in Figure 3.11. If one subtracts directly without considering the overlap, the overlap will have been subtracted twice. Calculating the overlap however becomes too difficult to consider for this thesis.

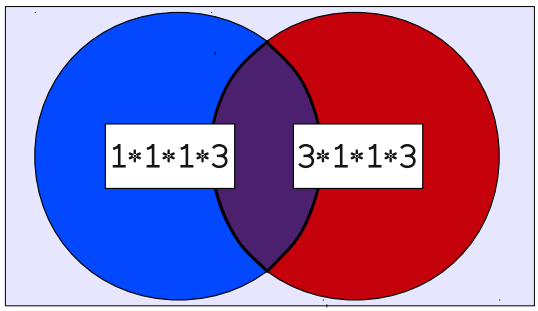

Figure 3.11: Venn diagram showing overlap between 2 subtractions

3.3.2 Perm Map

In order to find out if a given distribution of colors from a colormap is the correct one, the PermMap was devised. Its function is to try to satisfy all cells given its color from the colormap while not adding any unwanted cells. This is accomplished by defining that any new live cells added must be dead in the next generation (symbol "Dies Now" from Figure 3.5), this way we can add live cells to satisfy other cells and make them disappear so the configuration will still match the original. If successful it means the color distribution leads to a goal and the PermMap will give us the first valid solution it found.

The PermMap iterates over all cells until it finds a cell that cannot be satisfied or all cells are satisfied, in either case we are done. Figure 3.12 shows both cases.

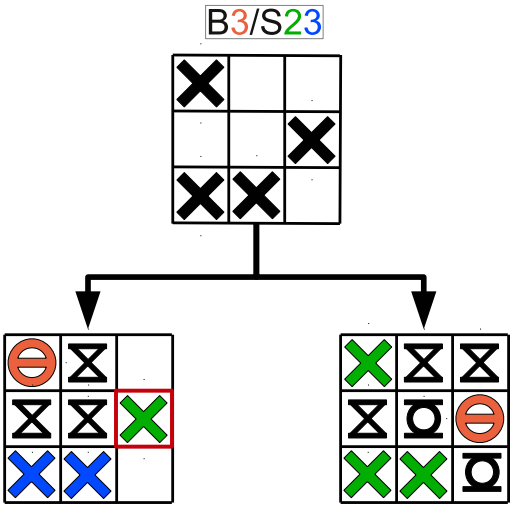

Figure 3.12: Two permMaps from the same configuration and rule. The left cannot satisy the highlited cell, the right has satisfied all cells

For each cell we check if it still has any hope of being satisfied, if not then we stop. If there is only one way to satisfy the cell, that 'way' is carried out. Figure 3.13 shows two such cases. In the first case the corner cell requires 3 cells, since there is exactly 3 cells available all of them must be live for it to be satisfied. The second case shows a cell that requires 4 neighbours while 5 exists, however one of them has previously been marked as finally dead meaning only 4/5 cells are available. With the requirement being 4 and the available being 4 there is only one way to satisfy it.

If no change happened in the last iteration (of all cells) and we didn't satisfy all cells we need to recursively check all options for satisfying the cells remaining. Figure 3.14 shows two cases where recursion is needed as there is more than one way to satisfy the cells. The first case shows a cell requiring 3 neighbours when 8 is available. To find how many combinations exist, and hence

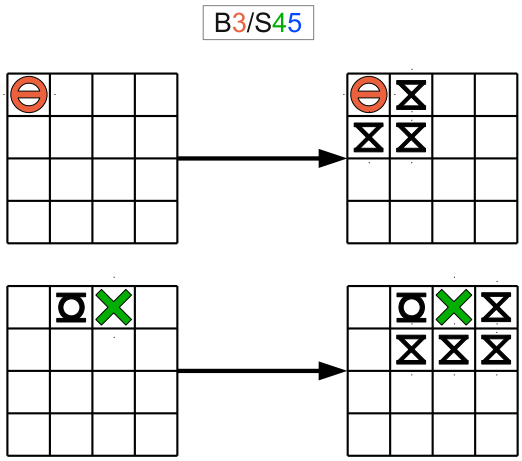

Figure 3.13: The cells in the left PermMaps can only be satisfied one way, so they can be solved directly

how many recursions must be made, the calculation is an unordered combinatorial problem[25]. There are $C_n^k, (k = 3, n = 8) => C_8^3 = \frac{8!}{3!(8-3)!} = 56$ combinations for the first case. In the second case the cell requires 4 when there is 5 available, this gives $C_5^4 = 6$ combinations.

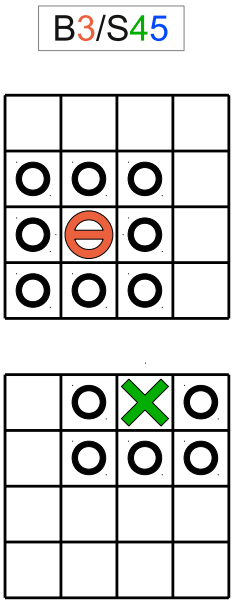

Figure 3.14: The cells can be satisfied in multiple ways, so recursion is needed

To recurse as little as possible we find the *smallest choice*, that is the cell that has the least possible ways to be satisfied. Figure 3.15 shows a 3x3 matrix where a choice must be made. The orange cell requires 1 from $3 = C_3^1 = 3$, while the green cell requires 2 from $5 = C_5^2 = 10$. This gives the *smallest choice* as the orange cell where only 3 forks are needed. We iterate over its ways to

be satisfied and recurse. If any of the forks find the solution, we are done.

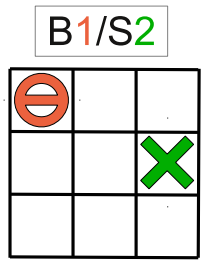

Figure 3.15: The orange cell requires 1 from 3, while the green cell requires 2 from 5.

Chapter 4

Results and Analysis

4.1 Reversible Cellular Automata

```
Original:
oooooooo
ooxoooxx
xoxxxoxx
xxxxxxox
xxoxxoox
xxooxooo
xoxoxoxx
ooooooxx

previous:
oxoxoxox
oooooooo
oxoxoxox
oooooooo
oxoxoxox
oooooooo
oxoxoxox
oooooooo

Arrived 1 generation(s) into the future.
Lowest population was 31 (0.0%).
Highest population was 31 (0.0%).


Current:
oxoxoxxo
oxxxxxxx
xxoxxxoo
xoooooxx
oxoxooxx
xoxoxxxx
xooooxo
oxoxoxxx
```

Figure 4.1: The Reverse CA player evolving an 8x8 matrix 1 step forward using a randomly generated *previous* 8x8 matrix. The rule used was B234/S234 with second-order Cellular Automata.

Using the code provided in the appendix we searched out configurations of various size, backtracked them and compressed them with the built in java deflate algorithm (zip) at each step. We found that we could easily move forward and backward through time while the configuration seemed to change drastically every step. We tried both Block cellular automata with the seeds and tron rules as well as second order CA with the B234S234 when trying to compress the data. The result of the compression however did not show much

```
Original:
xxxooxoxxx
oxooooxoox
xxooxxooxo
xooxxxoooo
xxoxxxxxox
oxxooxxxoo
xooxxoxooo
xoxooooxox
ooooxxxxox
xxxoooooxo

previous:
oxoxoxoxox
oooooooooo
oxoxoxoxox
oooooooooo
oxoxoxoxox
oooooooooo
oxoxoxoxox
oooooooooo
oxoxoxoxox
oooooooooo

Arrived 500 generation(s) into the future.
Lowest population was 33 (-31.25%).
Highest population was 65 (35.41666666666667%).

Current:
ooxxxxxxoo
xoxoooxxoo
xooxoxoooo
xxxoooxooo
xoxoxxoooo
xoooxxxxxx
oxooxoooxx
ooxxoxoxox
xoxxxoxooo
oooooxooxox
```

Figure 4.2: The Reverse CA player evolving a 10x10 matrix 500 steps forward using a randomly generated *previous* 10x10 matrix. The rule used was B234/S234 with second-order Cellular Automata. The algorithm took less than a second to finish.

```
Original:
oxoxoooxooox
xooxooxxxxox
oooxoxooxoxo
oxoooxxoxoxo
xoxxxoxooxox
xxooooooxooxo
ooooxoxxooxo
oooooxoxxxxx
ooooooxoxoxxo
xoooooxoxxoox
xxoxoooxoxox
xoooxooxxxxo

previous:
oxoxoxoxoxox
oooooooooooo
oxoxoxoxoxox
oooooooooooo
oxoxoxoxoxox
oooooooooooo
oxoxoxoxoxox
oooooooooooo
oxoxoxoxoxox
oooooooooooo
oxoxoxoxoxox
oooooooooooo

Runtime: 62
G: 3000, generations/sec :48387.096774193546

Runtime: 95
G: 6000, generations/sec :63157.89473684211

Runtime: 129
G: 9000, generations/sec :69767.44186046511

Arrived 10000 generation(s) into the future.
Lowest population was 47 (-25.396825396825395%).
Highest population was 93 (47.61904761904761%).

Current:
oooooooxxxoo
ooxooxxooxxx
xoxxxxooxxox
xooxxooxoxxx
xxoooxxoxxoo
xxoxxoxxoxox
oxoxooxxoxoo
oxxxxooooooo
xoxxooxoxoxo
xxoxoxoxxxxx
xoxoxoooxxoo
oxoxoooxooxx
```

Figure 4.3: The Reverse CA player evolving a 12x12 matrix 10000 steps forward using a randomly generated *previous* 12x12 matrix. The rule used was B135/S024 with second-order Cellular Automata. The algorithm took approximately 150ms to finish.

promise.

For block cellular automata it was found that since no information is lost[14], it will not compress data (or very little). The reason for this comes from the ability to move in both directions, meaning it does not change itself so that the path is lost.

When looking at second order CA there exists another problem; To move in any direction, 2 configurations are needed because the algorithm needs the configuration at both time t and t-1 to calculate t+1. This lead us to generate a random configuration as t-1 and set the data to be compress as t. We can then calculate t+1 and further.

However, this does not solve the problem, say that we found a configuration at t+23 that could be compressed to just 30% of the original. If we wanted to store this in place of the original we would also need to store the configuration at t+24 in order for the decompression algorithm to be able to get back to the original data (plus the number 23 to know how many steps to go). When we combine the size of t+23, which was small, with t+24 we see that the sum of these configurations must be less than the original. When testing this did never happen.

In order to combat the latter obstruction we investigated the possibility to store 2 consecutive configurations, but with a space between them. Although it would be possible to find the original, this would the require the decompression algorithm to work more as it first needs to find the missing part. In testing however, even this did not become smaller then the original.

Another problem we ran into with reversible CA was the fact that in finite space loops must appear. This can easily be observed with a thought experiment; Consider an algorithm that fills a 3x3 matrix with boolean values (true or false). There are $2^{3*3} = 512$ possible 3x3 matrices created by this algorithm. If the algorithm is run $512 + 1$ times, the algorithm must have created atleast 1 matrix that has already been created.

This applies to Cellular Automata also, in finite space there is a finite number of possible configurations. If the CA is run more times than the number of configurations then the CA has entered a loop. It is not known however when it entered a loop or how man cycles the loop has. Note that a loop could in theory contain all possible configurations in the given space, or even just 1 (normally called *still life*)[14].

4.2 Genetic Algorithm

The GACABacktracer was tested several times with different rules and configurations. All tested configurations was known to have at least one previous

configuration in the given rule(generated by running the CA forward one step on the input first). Here follows a sample run:

```
Our Origin:
ooxoxoxo
xoooxooo
xoooxoxo
oooooooo
oxxxxxxo
oxooooox
oxoooxxo
xooxxxxo
Points: 240
G: 10, generations/sec :4.504504504504504, T minus  8 seconds.
Entropy?: 3198082052.
Best: 199, 82.916664%, CurrentAverage: 83.1192
GA complete. 50 generations of CA has passed.
The search took  39 seconds.
Highest score was '210' points 87.5%
Here follows the best solution we found(G:2):
xxxooxox
oxoooxo
xxoxoxxx
xxooxoxo
xoxoxooo
ooxoxoxx
xxoooxox
xoxxxxoo
Size 4: 39315, 39315.0.
```

The GA did not manage to find a solution, but instead found a configuration that looks 87.5% similar. In this test the GA was given an initial population of 10'000 genes and 50 generations to evolve.

In the following test the GA was given 5000 generations to evolve:

```
Our Origin:
xoxxoxxx
oooxoxox
xxxoxxxo
xoxxxoox
xoxoxoxx
xoxooxox
xoxoxxoo
oooxxxoo
Points: 360
G: 10, generations/sec :37.3134328358209, T minus  2 minutes 13 seconds.
Entropy?: 31688012.
Best: 262, 72.77778%, CurrentAverage: 143.722
G: 100, generations/sec :40.96681687832856, T minus  1 minute 59 seconds.
Entropy?: 31830804.
Best: 284, 78.888885%, CurrentAverage: 143.088
G: 500, generations/sec :46.29629629629629, T minus  1 minute 37 seconds.
Entropy?: 31849362.
Best: 295, 81.94444%, CurrentAverage: 142.536
G: 1000, generations/sec :47.3215975771342, T minus  1 minute 24 seconds.
Entropy?: 31710974.
Best: 304, 84.44444%, CurrentAverage: 146.966
G: 1500, generations/sec :47.70233741453331, T minus  1 minute 13 seconds.
Entropy?: 31732474.
Best: 310, 86.11111%, CurrentAverage: 144.514
G: 2000, generations/sec :48.46604953230263, T minus  1 minute 1 second.
Entropy?: 22437848.
Best: 337, 93.61111%, CurrentAverage: 141.472
G: 2500, generations/sec :50.67293659802173, T minus  49 seconds.
Entropy?: 22749010.
Best: 338, 93.88889%, CurrentAverage: 143.889
G: 3000, generations/sec :52.24751388913078, T minus  38 seconds.
Entropy?: 22621502.
Best: 338, 93.88889%, CurrentAverage: 142.858
G: 3500, generations/sec :53.422064839123266, T minus  28 seconds.
Entropy?: 22425088.
Best: 338, 93.88889%, CurrentAverage: 146.376
G: 4000, generations/sec :54.328633906500414, T minus  18 seconds.
Entropy?: 22398090.
Best: 338, 93.88889%, CurrentAverage: 149.141
G: 4500, generations/sec :55.07214450930719, T minus  9 seconds.
Entropy?: 22364086.
Best: 338, 93.88889%, CurrentAverage: 137.321
GA complete. 5000 generations of CA has passed.
The search took  1 minute 29 seconds.
```

```
Highest score was '338' points 93.88889%
Here follows the best solution we found(G:2):
oxxxxxox
xxoxxxxx
oxxoxxxx
ooxxxoox
ooooxxxx
oxoxoxxx
xxoxxxox
ooxoooxo
Size 4: 89831, 89831.0.
```

Because one needs 100% for lossless compression, the number of generations was increased. Here another sample run where the number of generations to go was increased to 50'000 and the number of genes lowered to 1000:

```
Our Origin:
oxooooxo
oooxxooo
ooooxxoo
oooxxoox
oxxoxxxx
xoxoxoxx
xoxxooxx
xooxxxox
Points: 310
G: 10, generations/sec :32.154340836012864, T minus  25 minutes 54 seconds.
Entropy?: 14307600.
Best: 271, 87.41936%, CurrentAverage: 169.71
G: 100, generations/sec :40.95004095004095, T minus  20 minutes 18 seconds.
Entropy?: 2391818.
Best: 294, 94.83871%, CurrentAverage: 260.948
G: 500, generations/sec :46.01932811780948, T minus  17 minutes 55 seconds.
Entropy?: 1948048.
Best: 295, 95.161285%, CurrentAverage: 259.931
G: 1000, generations/sec :46.86694474387215, T minus  17 minutes 25 seconds.
Entropy?: 1960052.
Best: 295, 95.161285%, CurrentAverage: 262.902
G: 1500, generations/sec :47.04406460718206, T minus  17 minutes 10 seconds.
Entropy?: 2114146.
Best: 295, 95.161285%, CurrentAverage: 263.194
G: 2000, generations/sec :46.917519001595196, T minus  17 minutes 3 seconds.
Entropy?: 2076338.
Best: 295, 95.161285%, CurrentAverage: 262.519
G: 2500, generations/sec :46.9386605583823, T minus  16 minutes 51 seconds.
Entropy?: 2015608.
Best: 295, 95.161285%, CurrentAverage: 260.311
```

Note: the output was cut here.

```
GA complete. 50000 generations of CA has passed.
The search took  17 minutes 34 seconds.
Highest score was '295' points 95.161285%
Here follows the best solution we found(G:2):
xxoooxxo
xxxxxooo
oooxxxoo
ooxxxoxx
xxxxxxxo
xxxoxxxx
xoxxxxxx
ooxoxxox
Size 4: 1054957, 1054957.0.
Insertions: 0
```

The algorithm did not manage to attain 100%, but instead stuck to the 95% that was discovered in the beginning.

From the results described above we can see that so far the GA has not achieved the goal. Even though the algorithm can quickly search over large matrices, without a 100% match the result cannot be used as intended for compression.

4.2.1 Measures to increase exploration

In order to increase the amount of exploration of the algorithm a couple of measures were implemented.

Reinitiation

After the algorithm has run for a given length of time the population was wiped out and recreated as if restarting the algorithm, but keeping the best solution.

Increased mutation

By increasing the mutation rate of the algorithm the offspring will be more random, leading to less exploitation and more exploration.

4.3 Colouring Algorithm

A 2x2 matrix was backtracked 1 generation in rule B13/S024 as shown in **??**. The algorithm was able to list several configurations that lead to it and finished in 16 milliseconds on the test system.

Figure 4.5 shows the average time(milliseconds) to backtrace random configurations in Game of Life(B3/S23).

Under all tests run the algorithm was able to find the previous configuration, of any configuration and rule, if it exists. The time it took however grows increasingly by the configuration size as shown by Figure 4.5. Figure 4.10 graphs the data from the table and clearly shows increasing growth of average time taken to solve configurations with the same rule (Game of Life 'S2/B32') of different sizes.

In order to judge what type of complexity growth the algorithm has the data was also plotted in a logaritmic graph as shown in Figure 4.11.

Figure 4.12 shows the difference between the growth of brute force and the colouring algorithm. Brute force was defined as testing all possible matrices of the given size, using 0.2 ms to test each candidate. That is, both algorithms searching for all possible previous configurations. The graph clearly shows that the colouring algorithm grows slower.

We have successfully backtracked configurations of sizes 2, 3, 4, 5 and 6. The algorithm can handle most configurations of size < 4 in milliseconds while size 6 took 2 hours. The algorithm can take any configuration (provided it can be placed in a square matrix) as input as well as any binary CA rule.

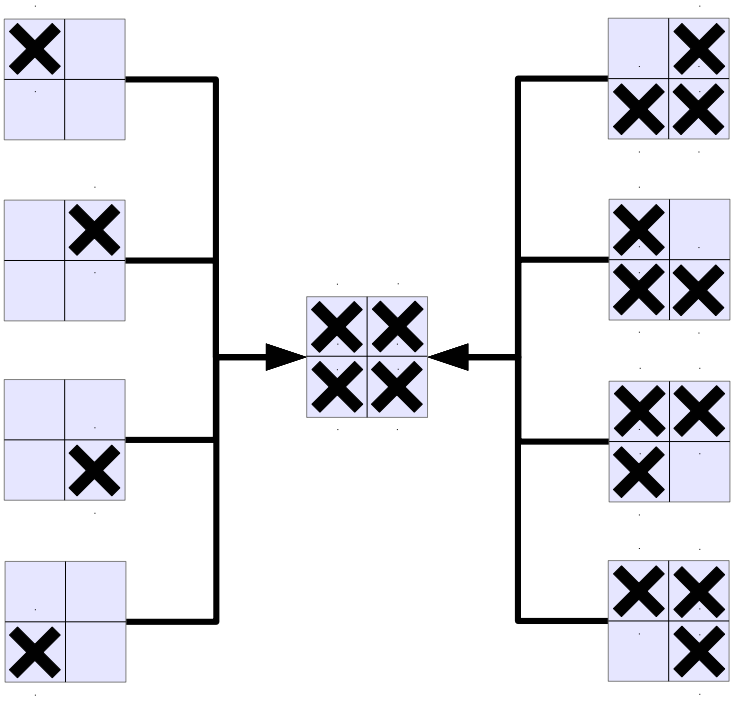

Figure 4.4: Backtracing a 2x2 matrix in the B13/S024 rule. The center configuration has 8 previous configurations leading to it in 1 generation. The algorithm took 17ms to finish.

size	2x2	3x3	4x4	5x5	6x6
time (ms)	6.5	31.9	1455	109866	7095285

Figure 4.5: Average time(in milliseconds) to backtrace matrices of different sizes in Game of Life (B3/S23) using the Colouring Algorithm.

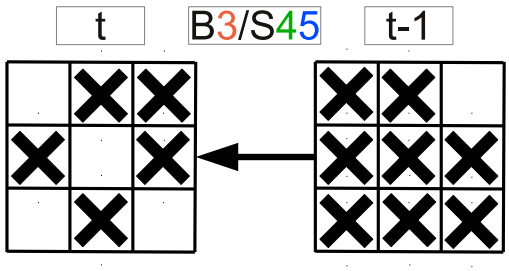

Figure 4.6: Backtracing a 3x3 matrix in the B3/S234 rule. The left configuration has exactly 1 configuration leading to it in 1 generation. The algorithm took 8ms to finish.

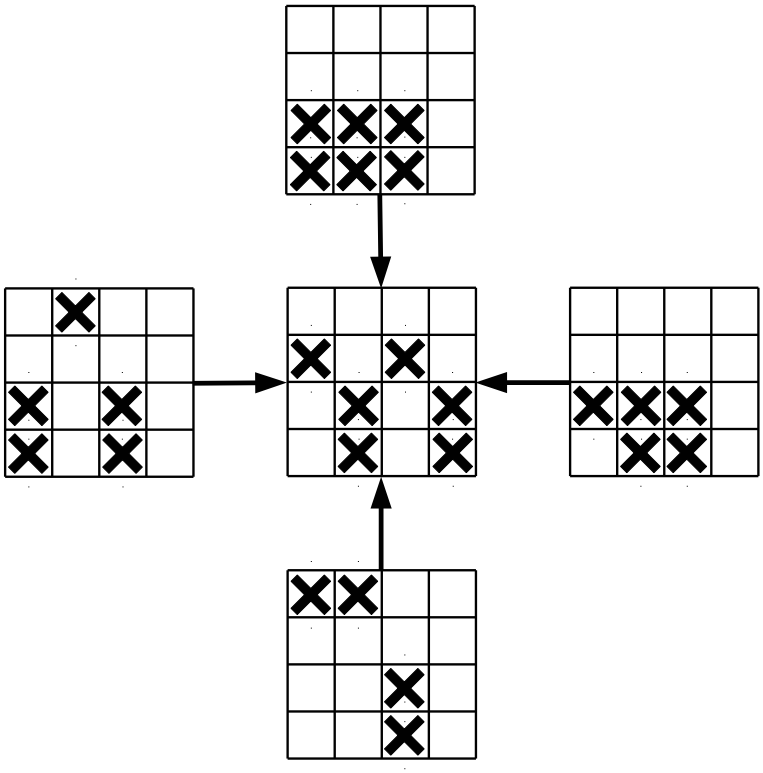

Figure 4.7: Backtracing a 4x4 matrix in the B24/S45 rule. The center configuration has 4 configuration leading to it in 1 generation. The algorithm took 255ms to finish.

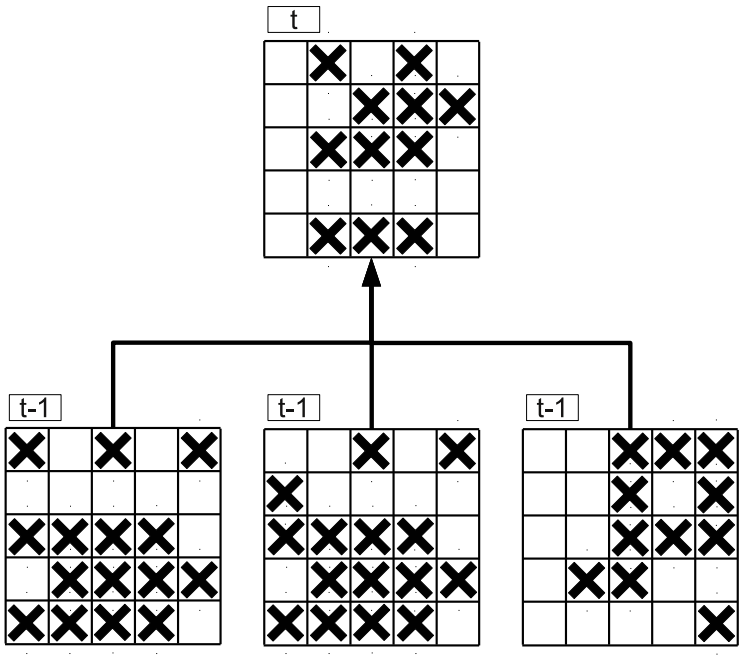

Figure 4.8: Backtracing a 5x5 matrix in the B248/S45 rule. The top configuration has 3 configuration leading to it in 1 generation. The algorithm took 36512ms to finish.

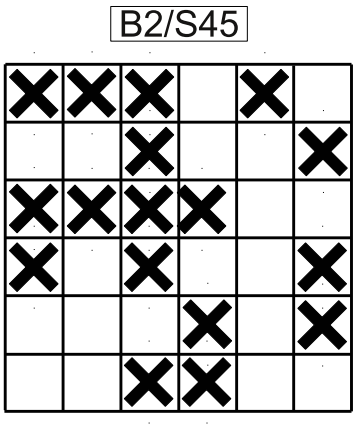

Figure 4.9: Backtracing a 6x6 matrix in the B248/S45 rule. This configuration has no parents, hence it is a Garden of Eden. The algorithm took 158660ms to finish.

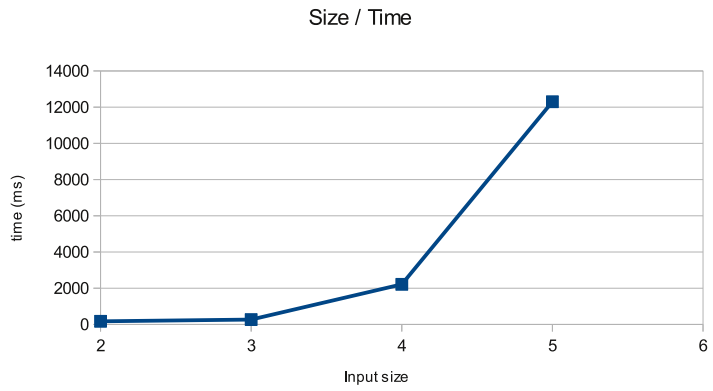

Figure 4.10: Graph of size vs time with the Colouring Algorithm, note the increasing growth.

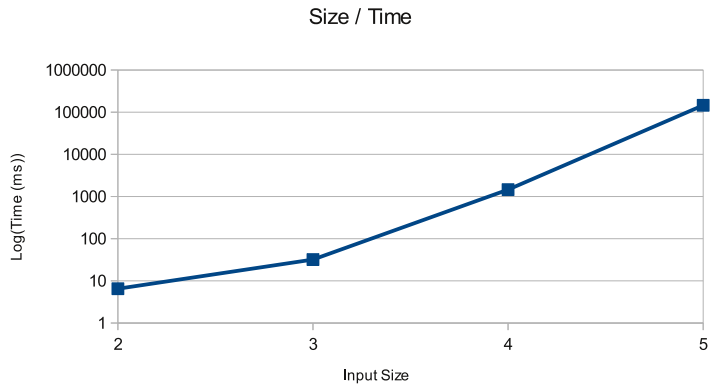

Figure 4.11: Logaritmic graph of size vs time with the Colouring Algorithm.

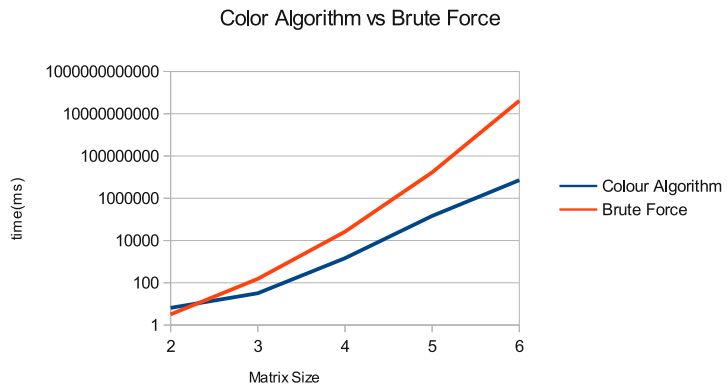

Figure 4.12: Logarithmic graph of size vs time comparing growth of Bruteforce and the Colouring Algorithm.

Figure 4.13 shows a full backtrack of a 4x4 configuration, not how most of the encountered configurations have at least one previous configuration. The initial configuration of Figure 4.13 has 22 configurations leading to it.

By inspection of the tree in Figure 4.13 the original matrix has 10 live of 16 cells. The configuration leading to it with the fewest live cells, highlighted in the figure, has only 5/16 live cells. The highest count of live cells is 11, which can also be seen as 5/16 dead cells.

The average number of cells in the predecessors in this tree is 9.181. Figure 4.14 shows a histogram of the number of different live cell counts observed in the tree from Figure 4.13. It is of importance to note that the histogram seems not to be a normal distribution. From the available data it seems that the number of live cells in the matrices do not obey any regulations, and therefore, is not predictable.

4.3.1 Measure to decrease complexity on large matrices

Because the goal is to compress data, the algorithm needs to be able to handle bigger matrices. As has been seen, a binary matrix of size 6 (the largest backtraced so far) contains only $6 * 6 = 36$ bits or $36/8 = 4 + 1/2$ byte. In order to accomplish handling bigger matrix sizes the matrix must be split into smaller parts that can be solved directly in order to reduce the overall complexity.

In order to test if the theory of splitting bigger matrices into smaller for faster solution holds a proof-of-concept was developed. First a 6x6 matrix was selected and backtraced directly. The matrix is given in Figure 4.15 and took 114 minutes to backtrace.

The matrix was then split into four 3x3 areas as shown in Figure 4.16. Each of these matrices where altered so that the borders that face eachoter was set as "don't care" cells, Figure 4.17 shows which cells of the North-West corner became "don't cares".

The "Don't care" cells affect other cells as normal, but their requirements are ignored by the algorithm. They are also set as dead if they were alive.

By then solving all the four 3x3 areas directly, all their solutions were recorded. By permutating the recombination of all the solutions the solutions for the original 6x6 matrix was sought.

Unfortunately, because the South-West corner was completely empty (no live cells) the algorithm did not return any solutions. This is because there is no Colors to iterate over, hence no Perm Maps being created or solved. It was therefore decided to increase the size of the South-West corner to 4x4.

By permutation and recombination of the solutions of the 4x4 and the three 3x3 matrices, 3/4 of the previous configurations for the original 6x6 matrix

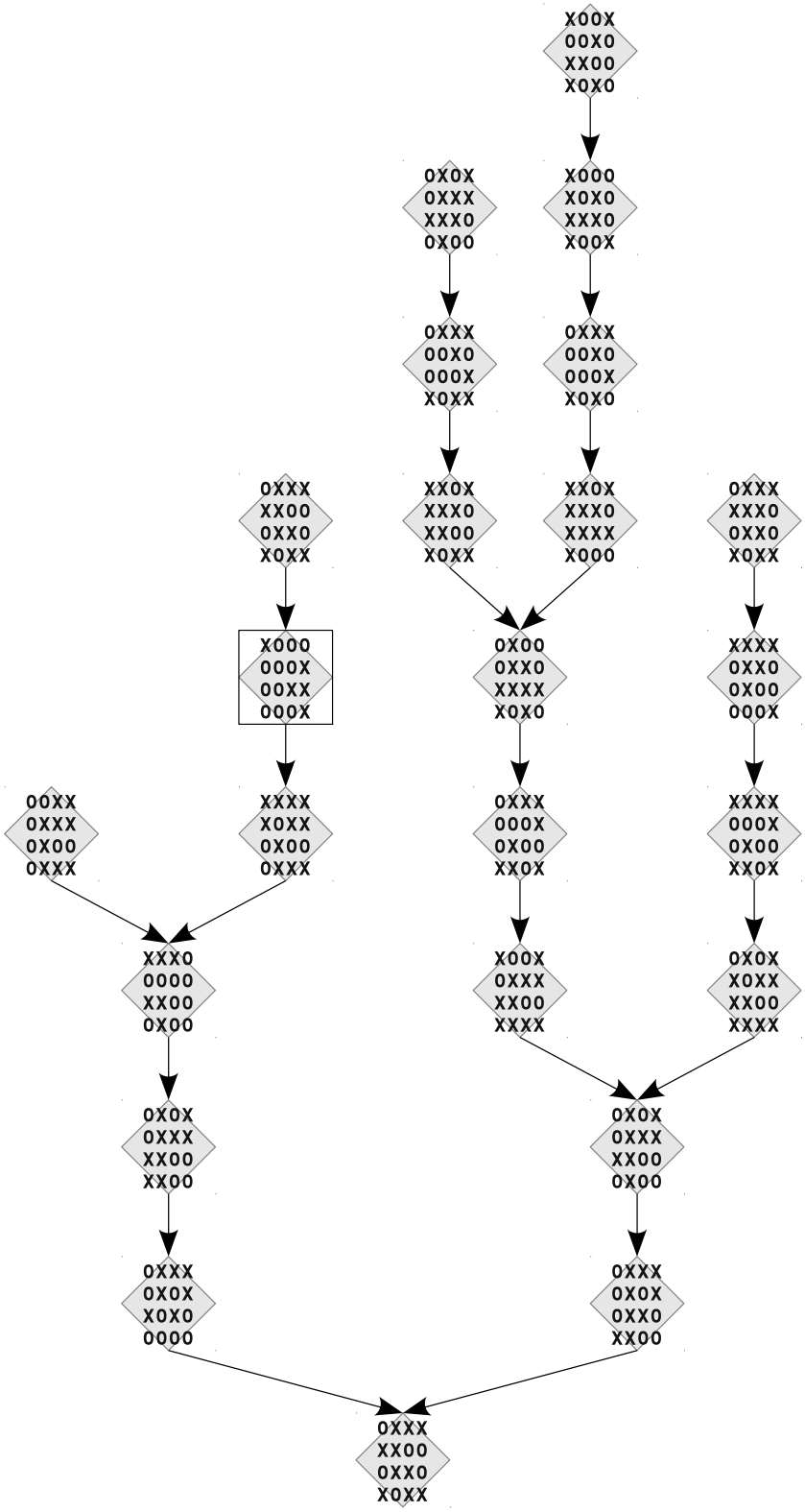

Figure 4.13: A full backtrace of a 4x4 configuration with rule B1358/S02467.

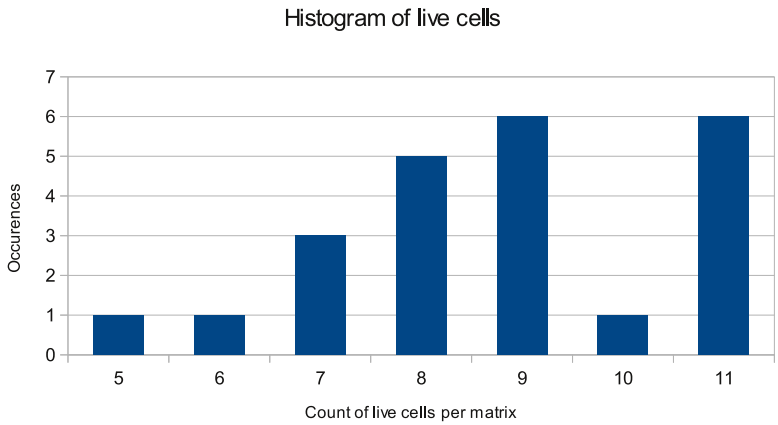

Figure 4.14: Histogram of the live cell counts

was found. Figure 4.18 shows the solutions that were found, along with the last solution that could not be found.

The time to solve each 3x3 matrix was less than a second, while the 4x4 took 3 seconds. The recombination process took 30 minutes.

Looking at the total time of 30 minutes compared with the original 2 hours the new timing is a positive result. The approach did however not yield all the previous configurations.

4.4 CA rules

In order to find out which rule is optimal for compressing data we have tested all the rules on matrixes on size 3 to find out which rule most often have a previous configuration (have the least gardens of eden). To test this we used to colour algorithm to test each rule on 1000 random matrices and check how many of the matrices had a solution for the given rule. This is an indication of how many gardens of Eden exist for the rule in 3x3 matrices, the result for each rule is given in a percentage of how many had a solution.

The rule that was found to be the best for size 3 is 'B7531/S76420' with 91%. 4.19 Shows the 10 best rules in this test. A longer list can be seen in appendix section 7.1.

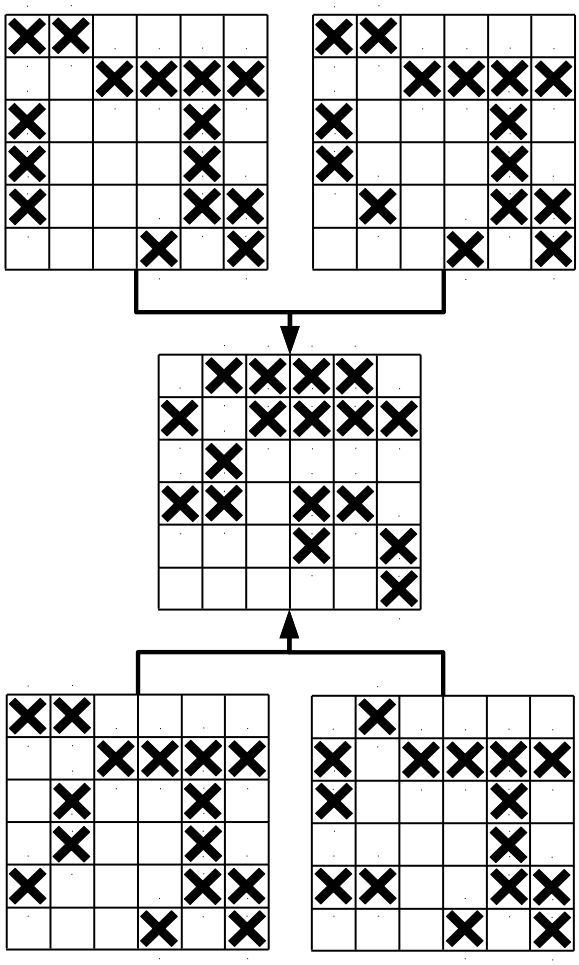

Figure 4.15: 6x6 matrix backtraced in Game of Life (B3/S23) using the Colouring algorithm. The center cell is the origin and has 4 previous configurations leading to it in 1 generation

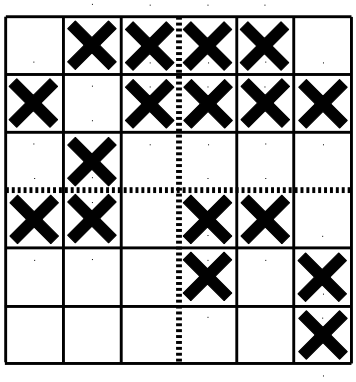

Figure 4.16: The 6x6 matrix split into four 3x3 areas

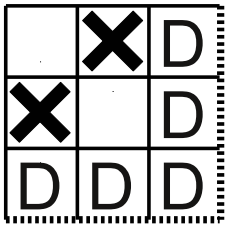

Figure 4.17: 3x3 North-West corner showing "don't cares" as D

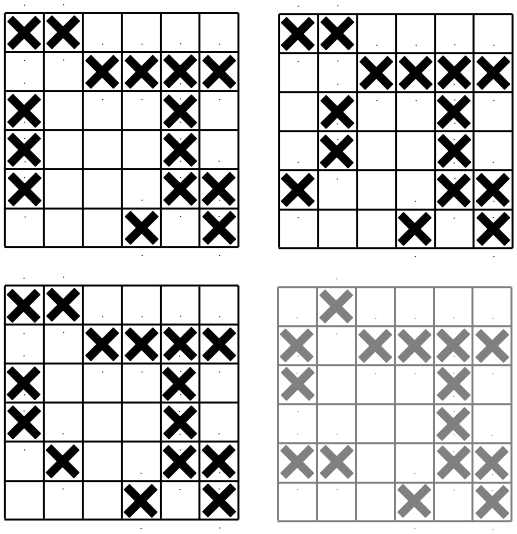

Figure 4.18: The solutions for the 6x6 matrix that were found by the split approach. The grayed out (lower right) solution could not be found

```
B531/S420 => 85%
B7531/S420 => 85%
B531/S7420 => 87%
B6531/S6420 => 87%
B7531/S7420 => 87%
B531/S6420 => 89%
B6531/S76420 => 89%
B7531/S6420 => 89%
B531/S76420 => 91%
B7531/S76420 => 91%
```

Figure 4.19: The 10 best rules selected by the test

Chapter 5

Discussion and Future Work

5.1 Discussion

5.1.1 Reversible Cellular Automata

Reversible Cellular Automata was implemented as both Second-Order and Block Cellular Automata with success. The algorithms were able to run in both directions taking input of varying size. Both algorithms were tested by attempting to compress the new configuration at each step.

Although the reversible Cellular Automata implementation was able to go both forth and back in time it did not compress the data. A measure was implemented to lower the threshold by allowing a *triplet*, storing two configurations of second-order ca with a 1 generation gap, may have worked partially. But ultimately did not allow for compression either.

The implementation of the Reversible Cellular Automata only considered Block Cellular Automata and Second-Order Cellular Automata in finite space. In infinite space the result could be different. The thesis has however proven that Cellular Automata must always enter a loop in finite space.

As a closing note on Reversible CA for Data Compression it is not clear how running such a model on some data affects its entropy. This is suggested for further study.

5.1.2 Genetic Algorithm

The Genetic Algorithm was implemented to search the CA space for configurations that lead to some given configuration. Two crossover functions and two mutation functions were implemented. The CA was able to perform a search in the CA space, leading to results of up to 95% similar to the original.

The genetic algorithm was able to find an answer for some 2x2 and 3x3 matrices, but in most cases it found a local optimum where the best answer is about 90% the same as the global maximum (a correct answer). Several

tweaks of the population size, the mutation rate and the mutation function were attempted but it only helped marginally.

The problem with the GA was that it it would in most cases get stuck with exploiting the current best answer instead of exploring for the global optimum. Even when changing the balance greatly toward exploration it would finally get stuck on exploiting something that didn't lead to a real answer. Some of the reason for this could be that there is not enough relation between CA configurations or that we could not produce mutation functions and recombination functions that found the relation.

The GA may however have application in lossy compression using CA, which could be studied further.

5.1.3 Coloring Algorithm

The colouring algorithm was designed, implemented and tested. The algorithm was able to identify the previous configurations of any configuration and rule it was given, when one existed. Sizes tested range from 2x2 to 6x6. On sizes smaller than 5x5 the algorithm will generally finish in milliseconds, while 6x6 matrices may take several hours.

Using the algorithm all binary CA rules in the Moore neighbourhood was tested on 3x3 matrices to rank them in order of how often Garden of Edens appear. Rules containing B8, B0 and rules with no birth were excluded as they seldom appear. The rule found to have the least amount of Gardens of Eden in Null Boundary 3x3 space was B1357/S02467, a list of the 1000 best can be found in appendix section 7.1.

Compressing actual file become very difficult however, this is mainly because of the rapid growth nature of the algorithm. So far the biggest matrices that have backtracked directly is 6x6, which takes from 160 seconds to 5 hours. Since each bit in a file would be treated as a cell, a small 20 KiB file would amass to a 405x405 matrix ($\sqrt{20 * 1024 * 8} = 405$). If the algorithm started backtracking such a matrix directly it wouldn't finish before our sun goes cold (about 5 billion years).

In order to compress data, the algorithm needs to work on matrices bigger than what the it can handle in a timely manner. By splitting the matrix into smaller parts that can be solved quickly, then recombining all their possibilities to find the solutions the problem may become smaller.

As was shown in subsection 4.3.1 this may hold merit. But is not implemented by the algorithm.

5.2 Future Work on Cellular Automaton based Data Compression

Optimizations and a real implementation of the idea of splitting the CA configuration into smaller parts and solving them independently could be considered for future study.

An important aspect of Cellular Automata that has not been considered in this thesis is multiple state CA, that is Cellular Automata with more than 2 states. Such CA could in theory pack more information per cell and thus more information in small matrices. The application of multiple state CA to data compression is therefore suggested as future work.

Chapter 6

Conclusion

The Genetic algorithm was tried and discarded as it could not find any previous configurations. Second-order cellular automata was discarded because storing a *way back* to the original would require storing 2 matrices by the same size as the original which did not result in smaller files in any of the tests. Block cellular Automata did not in any test make the matrix smaller.

A method to systematically search for previous configurations in 2 dimensional binary Cellular Automata was defined as the *Colouring Algorithm* and a proof-of-concept based on this definition was implemented and tested.

The algorithm works and it is better than a brute force approach, in all tests where a previous configuration was known to exist the algorithm found it. The algorithm cannot however directly achieve the goal of data compression because of its complexity growth. Although the original goal of data compression could not be achieved the colouring algorithm in itself is an achievement.

The bottom line about using Cellular Automata for lossless Data Compression is, however, that it seems very difficult.

Bibliography

[1] Michael J. Leamy. Application of cellular automata modeling to seismic elastodynamics. *International Journal of Solids and Structures*, 45 Issues 17:4835–4849, 2008.

[2] Sugata Mitra and Sujai Kumar. Fractal replication in time-manipulated one-dimensional cellular automata. *Complex Systems*, 16 Issues 3:191–207, 2006.

[3] David Salomon. *A Guide to DATA COMPRESSION METHODS*. Springer-Verlag New York Inc., 2002.

[4] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40 Issue 9:1098–1101, 1952.

[5] David Salomon. *Data compression: The Complete Reference*. Springer-Verlag New York Inc., 2004.

[6] C.E. Shannon and W. Weaver. *The mathematical theory of communication*. University of Illinois Press, Urbana, 1949.

[7] Marcelo J. Weinberger Ishai Amir, Eli Plotnik and Nir Tal. Comparison between universal data-compression algorithms applied to markov sources. In *Proceedings of the 17th Convention of Electrical and Electronics Engineers in Israel*, pages 111–114, 1991.

[8] Radii Radescu Adrian Traian Murgan. A comparison of algorithms for lossless data compression using the lempel-ziv-welch type methods. In *Proceedings of the IEEE-IMS Workshop on Information Theory and Statistics*, page 105, 1994.

[9] John von Neumann. The general and logical theory of automata. *L.A. Jeffress, ed., Cerebral Mechanisms in Behaviour*, The Hixon Symposium:1–31, 1951.

[10] Palash Sarkar. A brief history of cellular automata. *ACM Computing Surveys (CSUR)*, 32 Issue 1:80–107, 2000.

[11] Jarkko Kari. Reversible cellular automata. Technical report, University of Turku & University of Iowa, 2005.

[12] Martin Gardner. Mathematical games the fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[13] Andrew Trevorrow and Tom Rokicki. Golly, Cellular Automata Simulator. http://golly.sourceforge.net/. [Online; accessed 15-May-2012].

[14] Jarkko Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334 Issues 1-3:3–33, 2005.

[15] Stephen A. Silver. Life Lexicon. http://www.argentum.freeserve.co.uk/lex.htm. [Online; accessed 21-May-2012].

[16] Martin Gardner. *Wheels, life and other mathematical amusements*. W.H. Freeman & Company, 1985.

[17] Kees Kwekkeboom Marijn Heule, Christiaan Hartman and Alain Noels. Delft University of Technology: Smallest known Orphan. http://homepage.tudelft.nl/37b0y/. [Online; accessed 18-May-2012].

[18] Zhang Chuanwu. Cellular automata based m-sequence generation. In *Proceedings of the 8th International Conference on Electronic Measurement and Instruments*, pages 2.58–2.60, 2007.

[19] Jarkko Kari. Universal pattern generation by cellular automata. *Theoretical Computer Science*, 429:180–184, 2012.

[20] O.Slamet Santoso J. Santoso and B.Riyanto Trilaksono. Matrix characteristics for two dimensional nongroup cellular automata. In *Proceedings of the International Conference on Electrical Engineering and Informatics (ICEEI)*, pages 1–4, 2011.

[21] Olu Lafe. *Cellular Automata Transforms: Theory and Applications in Multimedia Compression, Encryption, and Modeling (Multimedia Systems and Applications)*. Springer-Verlag New York Inc., 2000.

[22] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[23] Stephen Marshland. *Machine Learning: An Algorithmic Perspective*. Chapman & Hall/CRC, 2009.

[24] Andrej Dobnikar Iztok Jeras. Algorithms for computing preimages of cellular automata configurations. Technical report, University of Ljubljana, 2007.

[25] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The art of computer programming*. Reading, Mass. : Addison-Wesley, 3rd edition, 1997.

Chapter 7

Appendix

7.1 CA rules tested on 3x3 matrices, 1000 best, rules with B8 excluded

```
   B6431/S7420 => 59%
 2 B6431/S7620 => 59%
   B6431/S76420 => 59%
 4 B64321/S => 59%
   B64321/S764 => 59%
 6 B64321/S7654 => 59%
   B6432/S41 => 59%
 8 B6432/S741 => 59%
   B6521/S430 => 59%
10 B6521/S5430 => 59%
   B6521/S6543 => 59%
12 B6521/S765430 => 59%
   B6531/S2 => 59%
14 B6531/S6421 => 59%
   B6531/S72 => 59%
16 B6531/S730 => 59%
   B6531/S75421 => 59%
18 B6531/S765421 => 59%
   B65321/S43 => 59%
20 B65321/S740 => 59%
   B65321/S764 => 59%
22 B6532/S75420 => 59%
   B6542/S5310 => 59%
24 B6542/S6532 => 59%
   B65431/S62 => 59%
26 B654321/S => 59%
   B654321/S74 => 59%
28 B65432/S76410 => 59%
   B721/S43 => 59%
30 B731/S642 => 59%
   B7321/S543 => 59%
32 B7321/S63 => 59%
   B7321/S643 => 59%
34 B7321/S6540 => 59%
   B7321/S6543 => 59%
36 B7321/S654 => 59%
   B7321/S730 => 59%
38 B7321/S754 => 59%
   B7321/S76543 => 59%
40 B7321/S7654 => 59%
   B732/S765420 => 59%
42 B7421/S31 => 59%
   B7421/S531 => 59%
44 B7421/S543 => 59%
   B7421/S7531 => 59%
46 B7421/S75430 => 59%
   B7421/S7543 => 59%
48 B742/S310 => 59%
   B742/S5431 => 59%
50 B742/S6532 => 59%
   B742/S75310 => 59%
52 B742/S7532 => 59%
   B742/S75431 => 59%
54 B7431/S5410 => 59%
   B7431/S6520 => 59%
56 B7431/S65410 => 59%
   B7431/S76420 => 59%
58 B7431/S765410 => 59%
```

B74321/S => 59%
B74321/S6530 => 59%
B74321/S654 => 59%
B74321/S73 => 59%
B74321/S763 => 59%
B74321/S764 => 59%
B74321/S7654 => 59%
B7432/S6410 => 59%
B7432/S765410 => 59%
B7521/S642 => 59%
B7521/S7430 => 59%
B7521/S7542 => 59%
B7521/S7642 => 59%
B7531/S2 => 59%
B7531/S72 => 59%
B75321/S4 => 59%
B75321/S => 59%
B75321/S64 => 59%
B75321/S6 => 59%
B75321/S740 => 59%
B75321/S74 => 59%
B75321/S7 => 59%
B75321/S7640 => 59%
B75321/S76 => 59%
B7532/S76420 => 59%
B7532/S7642 => 59%
B753/S5420 => 59%
B75421/S541 => 59%
B75421/S6541 => 59%
B75421/S6543 => 59%
B75421/S7631 => 59%
B75421/S763 => 59%
B75421/S76541 => 59%
B7542/S641 => 59%
B7542/S6532 => 59%
B7542/S7641 => 59%
B7542/S765320 => 59%
B75431/S72 => 59%
B75431/S762 => 59%
B75432/S541 => 59%
B75432/S641 => 59%
B75432/S6541 => 59%
B75432/S76541 => 59%
B7621/S743 => 59%
B7621/S76543 => 59%
B7631/S6320 => 59%
B7631/S642 => 59%
B7631/S6532 => 59%
B7631/S765320 => 59%
B76421/S631 => 59%
B76421/S6541 => 59%
B76421/S75430 => 59%
B76421/S7543 => 59%
B76421/S76543 => 59%
B7642/S5431 => 59%
B7642/S75430 => 59%
B7642/S75431 => 59%
B764321/S630 => 59%
B764321/S73 => 59%
B764321/S74 => 59%
B764321/S763 => 59%
B764321/S764 => 59%
B76521/S43 => 59%
B21/S76430 => 60%
B31/S7642 => 60%
B321/S3 => 60%
B321/S6540 => 60%
B321/S7654 => 60%
B32/S65420 => 60%
B421/S7541 => 60%
B421/S76541 => 60%
B421/S765430 => 60%
B421/S765431 => 60%
B42/S310 => 60%
B42/S65310 => 60%
B42/S7310 => 60%
B42/S7532 => 60%
B42/S765430 => 60%
B431/S7420 => 60%
B4321/S530 => 60%
B4321/S640 => 60%
B4321/S7530 => 60%
B4321/S7630 => 60%
B432/S75410 => 60%
B432/S76410 => 60%
B521/S643 => 60%
B521/S65430 => 60%
B531/S5421 => 60%
B5321/S30 => 60%
B5321/S540 => 60%

B5321/S6530 => 60%
150 B5321/S764 => 60%
B532/S65420 => 60%
152 B53/S765420 => 60%
B5421/S42 => 60%
154 B5421/S742 => 60%
B542/S765320 => 60%
156 B542/S76532 => 60%
B5431/S742 => 60%
158 B54321/S730 => 60%
B5432/S741 => 60%
160 B621/S65430 => 60%
B621/S765430 => 60%
162 B631/S642 => 60%
B631/S765320 => 60%
164 B6321/S653 => 60%
B6321/S763 => 60%
166 B6321/S7653 => 60%
B6421/S6431 => 60%
168 B6421/S6543 => 60%
B6421/S731 => 60%
170 B6421/S765430 => 60%
B6421/S76543 => 60%
172 B642/S7430 => 60%
B6431/S620 => 60%
174 B6431/S720 => 60%
B64321/S30 => 60%
176 B64321/S7630 => 60%
B651/S4320 => 60%
178 B6521/S6542 => 60%
B6521/S7542 => 60%
180 B6521/S75430 => 60%
B6531/S421 => 60%
182 B6531/S6532 => 60%
B6531/S7421 => 60%
184 B6531/S76421 => 60%
B6531/S76532 => 60%
186 B65321/S40 => 60%
B65321/S5 => 60%
188 B65321/S75 => 60%
B6532/S765420 => 60%
190 B65421/S52 => 60%
B65421/S752 => 60%
192 B65421/S7531 => 60%
B6542/S65320 => 60%
194 B6542/S765320 => 60%
B6542/S76532 => 60%
196 B65431/S52 => 60%
B65431/S652 => 60%
198 B65431/S742 => 60%
B65431/S752 => 60%
200 B65431/S7652 => 60%
B65432/S741 => 60%
202 B731/S742 => 60%
B732/S75420 => 60%
204 B7421/S541 => 60%
B7421/S65430 => 60%
206 B7421/S65431 => 60%
B7421/S731 => 60%
208 B7421/S7541 => 60%
B7421/S76541 => 60%
210 B7421/S765430 => 60%
B7421/S765431 => 60%
212 B742/S430 => 60%
B742/S532 => 60%
214 B742/S65310 => 60%
B742/S7310 => 60%
216 B742/S765430 => 60%
B7431/S620 => 60%
218 B7431/S6420 => 60%
B7431/S7420 => 60%
220 B7431/S7520 => 60%
B7431/S76520 => 60%
222 B74321/S640 => 60%
B74321/S7 => 60%
224 B74321/S7630 => 60%
B74321/S76530 => 60%
226 B7432/S5410 => 60%
B7432/S65410 => 60%
228 B7432/S76410 => 60%
B7521/S542 => 60%
230 B7521/S643 => 60%
B7521/S6542 => 60%
232 B7521/S7643 => 60%
B75321/S54 => 60%
234 B75321/S5 => 60%
B75321/S6530 => 60%
236 B75321/S764 => 60%
B75321/S76530 => 60%
238 B7532/S75420 => 60%

B7532/S765420 => 60%
240 B753/S65420 => 60%
B75421/S42 => 60%
242 B75421/S631 => 60%
B75421/S63 => 60%
244 B75421/S731 => 60%
B75421/S742 => 60%
246 B7542/S532 => 60%
B7542/S631 => 60%
248 B7542/S75320 => 60%
B7542/S7532 => 60%
250 B75431/S652 => 60%
B75431/S742 => 60%
252 B754321/S630 => 60%
B754321/S65 => 60%
254 B75432/S76410 => 60%
B7621/S430 => 60%
256 B7621/S65430 => 60%
B7621/S765430 => 60%
258 B7631/S742 => 60%
B76321/S653 => 60%
260 B76321/S7653 => 60%
B76421/S431 => 60%
262 B76421/S65430 => 60%
B76421/S6543 => 60%
264 B76421/S7431 => 60%
B76421/S7631 => 60%
266 B76421/S765430 => 60%
B7642/S5430 => 60%
268 B7642/S765310 => 60%
B76431/S6420 => 60%
270 B76431/S7420 => 60%
B76431/S76420 => 60%
272 B764321/S6 => 60%
B764321/S76 => 60%
274 B76432/S765410 => 60%
B7651/S4320 => 60%
276 B76521/S430 => 60%
B21/S6430 => 61%
278 B321/S73 => 61%
B32/S765420 => 61%
280 B421/S6541 => 61%
B421/S6543 => 61%
282 B421/S76543 => 61%
B42/S532 => 61%
284 B42/S6532 => 61%
B42/S65430 => 61%
286 B431/S7620 => 61%
B431/S76520 => 61%
288 B4321/S30 => 61%
B4321/S7 => 61%
290 B432/S410 => 61%
B432/S5410 => 61%
292 B432/S7410 => 61%
B521/S6542 => 61%
294 B521/S7542 => 61%
B521/S7643 => 61%
296 B531/S765421 => 61%
B5321/S43 => 61%
298 B5321/S5 => 61%
B5321/S730 => 61%
300 B5321/S743 => 61%
B5321/S754 => 61%
302 B5321/S75 => 61%
B532/S75420 => 61%
304 B53/S5420 => 61%
B53/S65420 => 61%
306 B5421/S531 => 61%
B5421/S743 => 61%
308 B5421/S7531 => 61%
B542/S75320 => 61%
310 B5431/S52 => 61%
B5431/S752 => 61%
312 B5431/S7652 => 61%
B54321/S4 => 61%
314 B54321/S => 61%
B5432/S41 => 61%
316 B5432/S5410 => 61%
B5432/S6410 => 61%
318 B5432/S75410 => 61%
B621/S76430 => 61%
320 B631/S5320 => 61%
B631/S65320 => 61%
322 B631/S742 => 61%
B6321/S3 => 61%
324 B6321/S53 => 61%
B6321/S753 => 61%
326 B6421/S76431 => 61%
B642/S765430 => 61%
328 B6432/S765410 => 61%

B6521/S643 => 61%
330 B6521/S65430 => 61%
B6521/S7643 => 61%
332 B6531/S520 => 61%
B6531/S7620 => 61%
334 B65321/S30 => 61%
B65321/S6530 => 61%
336 B65321/S730 => 61%
B65321/S7640 => 61%
338 B65321/S7643 => 61%
B65321/S76530 => 61%
340 B65421/S43 => 61%
B65421/S531 => 61%
342 B6542/S5320 => 61%
B6542/S7631 => 61%
344 B654321/S30 => 61%
B654321/S7 => 61%
346 B654321/S76 => 61%
B65432/S41 => 61%
348 B65432/S6410 => 61%
B65432/S7410 => 61%
350 B731/S7642 => 61%
B7321/S763 => 61%
352 B7421/S64310 => 61%
B7421/S6543 => 61%
354 B7421/S764310 => 61%
B7421/S76543 => 61%
356 B742/S5310 => 61%
B742/S7430 => 61%
358 B742/S75320 => 61%
B742/S765320 => 61%
360 B7431/S720 => 61%
B7431/S7620 => 61%
362 B74321/S30 => 61%
B74321/S530 => 61%
364 B74321/S7530 => 61%
B74321/S76 => 61%
366 B7432/S7410 => 61%
B7432/S75410 => 61%
368 B7521/S6430 => 61%
B7521/S76430 => 61%
370 B7531/S75421 => 61%
B7531/S765421 => 61%
372 B75321/S30 => 61%
B75321/S40 => 61%
374 B75321/S43 => 61%
B75321/S540 => 61%
376 B75321/S640 => 61%
B75321/S743 => 61%
378 B75321/S7530 => 61%
B75321/S7540 => 61%
380 B75321/S754 => 61%
B75321/S75 => 61%
382 B75421/S31 => 61%
B75421/S6531 => 61%
384 B75421/S76531 => 61%
B7542/S5320 => 61%
386 B75431/S52 => 61%
B75431/S62 => 61%
388 B75431/S752 => 61%
B75431/S7652 => 61%
390 B754321/S4 => 61%
B754321/S => 61%
392 B754321/S730 => 61%
B754321/S764 => 61%
394 B754321/S765 => 61%
B75432/S41 => 61%
396 B75432/S5410 => 61%
B75432/S6410 => 61%
398 B75432/S741 => 61%
B75432/S75410 => 61%
400 B7621/S43 => 61%
B7631/S65320 => 61%
402 B76321/S53 => 61%
B76321/S730 => 61%
404 B76321/S753 => 61%
B76421/S531 => 61%
406 B7642/S75310 => 61%
B7642/S765320 => 61%
408 B764321/S => 61%
B764321/S7 => 61%
410 B764321/S7630 => 61%
B76432/S65410 => 61%
412 B21/S7643 => 62%
B32/S75420 => 62%
414 B421/S6431 => 62%
B421/S7430 => 62%
416 B421/S743 => 62%
B421/S76431 => 62%
418 B42/S5310 => 62%

B42/S75320 => 62%
420 B431/S620 => 62%
B431/S720 => 62%
422 B4321/S730 => 62%
B4321/S7640 => 62%
424 B4321/S76 => 62%
B521/S6430 => 62%
426 B521/S76430 => 62%
B531/S75421 => 62%
428 B531/S76421 => 62%
B5321/S530 => 62%
430 B5321/S54 => 62%
B5321/S63 => 62%
432 B5321/S7530 => 62%
B532/S765420 => 62%
434 B5421/S43 => 62%
B542/S5320 => 62%
436 B542/S532 => 62%
B542/S731 => 62%
438 B542/S7532 => 62%
B5431/S652 => 62%
440 B54321/S5 => 62%
B54321/S764 => 62%
442 B5432/S65410 => 62%
B5432/S7410 => 62%
444 B5432/S765410 => 62%
B621/S7643 => 62%
446 B631/S5420 => 62%
B631/S75320 => 62%
448 B631/S7642 => 62%
B6421/S430 => 62%
450 B6421/S7430 => 62%
B642/S310 => 62%
452 B642/S65310 => 62%
B642/S65430 => 62%
454 B642/S75310 => 62%
B642/S75320 => 62%
456 B642/S76532 => 62%
B64321/S6 => 62%
458 B64321/S7 => 62%
B64321/S76 => 62%
460 B6432/S5410 => 62%
B6432/S6410 => 62%
462 B6432/S65410 => 62%
B6432/S76410 => 62%
464 B6521/S542 => 62%
B6521/S6430 => 62%
466 B6531/S7520 => 62%
B65321/S640 => 62%
468 B65321/S643 => 62%
B65321/S6543 => 62%
470 B65321/S76543 => 62%
B65421/S743 => 62%
472 B6542/S532 => 62%
B6542/S631 => 62%
474 B6542/S75320 => 62%
B6542/S7532 => 62%
476 B65431/S75420 => 62%
B654321/S75 => 62%
478 B721/S76430 => 62%
B7421/S6541 => 62%
480 B742/S5320 => 62%
B742/S65320 => 62%
482 B742/S65430 => 62%
B74321/S6 => 62%
484 B7432/S410 => 62%
B7531/S6421 => 62%
486 B7531/S7421 => 62%
B7531/S7520 => 62%
488 B7531/S76520 => 62%
B75321/S530 => 62%
490 B75321/S654 => 62%
B75321/S7654 => 62%
492 B7542/S731 => 62%
B754321/S7 => 62%
494 B75432/S410 => 62%
B75432/S65410 => 62%
496 B75432/S7410 => 62%
B75432/S765410 => 62%
498 B7631/S5320 => 62%
B7631/S5420 => 62%
500 B7631/S75320 => 62%
B7631/S7642 => 62%
502 B76321/S63 => 62%
B76421/S31 => 62%
504 B76421/S7531 => 62%
B7642/S310 => 62%
506 B7642/S430 => 62%
B7642/S5320 => 62%
508 B7642/S65310 => 62%

```
    B7642/S65320 => 62%
510 B7642/S6532 => 62%
    B7642/S7430 => 62%
512 B7642/S75320 => 62%
    B7642/S7532 => 62%
514 B7642/S76532 => 62%
    B7642/S765430 => 62%
516 B764321/S30 => 62%
    B76432/S6410 => 62%
518 B76432/S75410 => 62%
    B76432/S76410 => 62%
520 B21/S643 => 63%
    B421/S430 => 63%
522 B421/S43 => 63%
    B42/S5320 => 63%
524 B42/S631 => 63%
    B42/S6430 => 63%
526 B42/S65320 => 63%
    B42/S7631 => 63%
528 B42/S765320 => 63%
    B4321/S0 => 63%
530 B4321/S6 => 63%
    B521/S542 => 63%
532 B531/S421 => 63%
    B531/S6421 => 63%
534 B531/S7421 => 63%
    B531/S76520 => 63%
536 B5321/S3 => 63%
    B5321/S53 => 63%
538 B5321/S643 => 63%
    B5321/S654 => 63%
540 B5321/S763 => 63%
    B5321/S7654 => 63%
542 B5431/S642 => 63%
    B5431/S75420 => 63%
544 B5431/S7642 => 63%
    B54321/S30 => 63%
546 B54321/S6 => 63%
    B54321/S75 => 63%
548 B54321/S7 => 63%
    B5432/S410 => 63%
550 B621/S6430 => 63%
    B6421/S743 => 63%
552 B642/S5310 => 63%
    B642/S6532 => 63%
554 B642/S7310 => 63%
    B642/S7532 => 63%
556 B642/S765320 => 63%
    B64321/S730 => 63%
558 B6432/S75410 => 63%
    B6521/S76430 => 63%
560 B6531/S620 => 63%
    B65321/S530 => 63%
562 B65321/S543 => 63%
    B65321/S7530 => 63%
564 B65321/S7543 => 63%
    B65431/S420 => 63%
566 B65431/S642 => 63%
    B65431/S7642 => 63%
568 B65431/S765420 => 63%
    B654321/S540 => 63%
570 B654321/S5 => 63%
    B654321/S76540 => 63%
572 B65432/S410 => 63%
    B721/S7643 => 63%
574 B7321/S3 => 63%
    B742/S631 => 63%
576 B742/S7631 => 63%
    B74321/S40 => 63%
578 B74321/S730 => 63%
    B74321/S7640 => 63%
580 B7531/S421 => 63%
    B7531/S520 => 63%
582 B7531/S6520 => 63%
    B7531/S76421 => 63%
584 B75321/S6540 => 63%
    B75321/S730 => 63%
586 B75321/S76540 => 63%
    B75421/S43 => 63%
588 B75421/S743 => 63%
    B75431/S420 => 63%
590 B75431/S75420 => 63%
    B754321/S74 => 63%
592 B76421/S6431 => 63%
    B76421/S731 => 63%
594 B76421/S76431 => 63%
    B7642/S5310 => 63%
596 B7642/S532 => 63%
    B7642/S65430 => 63%
598 B7642/S7310 => 63%
```

```
    B76432/S5410 => 63%
600 B4321/S40 => 64%
    B4321/S6540 => 64%
602 B4321/S70 => 64%
    B4321/S76540 => 64%
604 B531/S520 => 64%
    B531/S6520 => 64%
606 B531/S7520 => 64%
    B531/S7620 => 64%
608 B5321/S653 => 64%
    B5321/S6540 => 64%
610 B5321/S6543 => 64%
    B5321/S73 => 64%
612 B5321/S753 => 64%
    B5321/S7643 => 64%
614 B5321/S7653 => 64%
    B5321/S76540 => 64%
616 B5321/S76543 => 64%
    B542/S31 => 64%
618 B5431/S420 => 64%
    B5431/S765420 => 64%
620 B54321/S74 => 64%
    B54321/S76 => 64%
622 B621/S643 => 64%
    B631/S65420 => 64%
624 B631/S75420 => 64%
    B6321/S73 => 64%
626 B6421/S43 => 64%
    B642/S5320 => 64%
628 B642/S532 => 64%
    B642/S6430 => 64%
630 B642/S65320 => 64%
    B6432/S410 => 64%
632 B6432/S7410 => 64%
    B6531/S20 => 64%
634 B65321/S63 => 64%
    B65421/S643 => 64%
636 B65421/S7643 => 64%
    B6542/S731 => 64%
638 B65431/S5420 => 64%
    B654321/S6540 => 64%
640 B654321/S7540 => 64%
    B721/S6430 => 64%
642 B721/S643 => 64%
    B7421/S7430 => 64%
644 B74321/S0 => 64%
    B74321/S6540 => 64%
646 B74321/S70 => 64%
    B74321/S76540 => 64%
648 B75321/S53 => 64%
    B75321/S643 => 64%
650 B75321/S653 => 64%
    B75321/S753 => 64%
652 B75321/S7643 => 64%
    B75321/S7653 => 64%
654 B75321/S76543 => 64%
    B75421/S7531 => 64%
656 B7542/S31 => 64%
    B75431/S642 => 64%
658 B75431/S7642 => 64%
    B75431/S765420 => 64%
660 B754321/S30 => 64%
    B754321/S5 => 64%
662 B754321/S6 => 64%
    B754321/S75 => 64%
664 B754321/S76 => 64%
    B7621/S76430 => 64%
666 B7621/S7643 => 64%
    B76321/S763 => 64%
668 B76421/S7430 => 64%
    B76421/S743 => 64%
670 B764321/S730 => 64%
    B76432/S410 => 64%
672 B76432/S7410 => 64%
    B42/S76430 => 65%
674 B4321/S7540 => 65%
    B5321/S543 => 65%
676 B5321/S7543 => 65%
    B542/S6531 => 65%
678 B542/S76531 => 65%
    B5431/S5420 => 65%
680 B5431/S7420 => 65%
    B642/S631 => 65%
682 B64321/S640 => 65%
    B64321/S7640 => 65%
684 B6531/S720 => 65%
    B65321/S3 => 65%
686 B65321/S653 => 65%
    B65321/S753 => 65%
688 B65321/S763 => 65%
```

```
    B65321/S7653 => 65%
690 B65431/S65420 => 65%
    B65431/S7420 => 65%
692 B731/S5420 => 65%
    B7321/S73 => 65%
694 B7421/S6431 => 65%
    B7421/S743 => 65%
696 B7421/S76431 => 65%
    B742/S6430 => 65%
698 B7531/S7620 => 65%
    B75321/S543 => 65%
700 B75321/S63 => 65%
    B75321/S6543 => 65%
702 B75321/S7543 => 65%
    B75321/S763 => 65%
704 B75421/S531 => 65%
    B7542/S6531 => 65%
706 B75431/S7420 => 65%
    B7621/S6430 => 65%
708 B7631/S65420 => 65%
    B7631/S75420 => 65%
710 B76321/S3 => 65%
    B76421/S430 => 65%
712 B76421/S43 => 65%
    B7642/S6430 => 65%
714 B764321/S640 => 65%
    B764321/S7640 => 65%
716 B31/S5420 => 66%
    B4321/S740 => 66%
718 B531/S20 => 66%
    B531/S42 => 66%
720 B531/S620 => 66%
    B531/S720 => 66%
722 B5421/S643 => 66%
    B5421/S7643 => 66%
724 B5431/S65420 => 66%
    B642/S7631 => 66%
726 B642/S76430 => 66%
    B64321/S0 => 66%
728 B6531/S42 => 66%
    B6531/S542 => 66%
730 B6531/S642 => 66%
    B6531/S7542 => 66%
732 B65321/S53 => 66%
    B6542/S31 => 66%
734 B654321/S640 => 66%
    B654321/S7640 => 66%
736 B654321/S7650 => 66%
    B7421/S430 => 66%
738 B7421/S43 => 66%
    B742/S76430 => 66%
740 B74321/S60 => 66%
    B74321/S740 => 66%
742 B74321/S7540 => 66%
    B7531/S42 => 66%
744 B75421/S643 => 66%
    B75421/S7643 => 66%
746 B7542/S76531 => 66%
    B75431/S5420 => 66%
748 B75431/S65420 => 66%
    B7621/S643 => 66%
750 B7642/S631 => 66%
    B7642/S7631 => 66%
752 B7642/S76430 => 66%
    B31/S75420 => 67%
754 B4321/S540 => 67%
    B4321/S60 => 67%
756 B4321/S75 => 67%
    B531/S542 => 67%
758 B531/S642 => 67%
    B531/S7542 => 67%
760 B542/S531 => 67%
    B542/S7531 => 67%
762 B631/S765420 => 67%
    B64321/S40 => 67%
764 B64321/S5 => 67%
    B64321/S75 => 67%
766 B65321/S73 => 67%
    B6542/S6531 => 67%
768 B65431/S20 => 67%
    B65431/S6420 => 67%
770 B65431/S76420 => 67%
    B654321/S650 => 67%
772 B731/S65420 => 67%
    B731/S75420 => 67%
774 B742/S31 => 67%
    B74321/S540 => 67%
776 B74321/S760 => 67%
    B7531/S542 => 67%
778 B7531/S620 => 67%
```

B7531/S642 => 67%
780 B7531/S7542 => 67%
B75321/S3 => 67%
782 B7542/S531 => 67%
B7542/S7531 => 67%
784 B754321/S640 => 67%
B7631/S765420 => 67%
786 B76321/S73 => 67%
B764321/S0 => 67%
788 B764321/S70 => 67%
B764321/S740 => 67%
790 B764321/S75 => 67%
B764321/S76540 => 67%
792 B31/S65420 => 68%
B421/S6430 => 68%
794 B421/S76430 => 68%
B42/S31 => 68%
796 B42/S731 => 68%
B4321/S5 => 68%
798 B4321/S760 => 68%
B531/S742 => 68%
800 B54321/S540 => 68%
B54321/S640 => 68%
802 B54321/S650 => 68%
B54321/S6540 => 68%
804 B54321/S7650 => 68%
B54321/S76540 => 68%
806 B6421/S76430 => 68%
B64321/S50 => 68%
808 B64321/S6540 => 68%
B64321/S70 => 68%
810 B64321/S740 => 68%
B64321/S76540 => 68%
812 B6542/S76531 => 68%
B65431/S720 => 68%
814 B654321/S740 => 68%
B742/S731 => 68%
816 B74321/S5 => 68%
B7531/S20 => 68%
818 B7531/S720 => 68%
B75321/S73 => 68%
820 B764321/S40 => 68%
B764321/S5 => 68%
822 B764321/S60 => 68%
B764321/S6540 => 68%
824 B764321/S760 => 68%
B764321/S765 => 68%
826 B421/S7643 => 69%
B4321/S50 => 69%
828 B5431/S6420 => 69%
B5431/S76420 => 69%
830 B54321/S7540 => 69%
B54321/S7640 => 69%
832 B6421/S6430 => 69%
B6421/S7643 => 69%
834 B64321/S60 => 69%
B64321/S750 => 69%
836 B64321/S760 => 69%
B6531/S6542 => 69%
838 B6531/S742 => 69%
B6531/S7642 => 69%
840 B6531/S76542 => 69%
B6542/S531 => 69%
842 B6542/S7531 => 69%
B65431/S7620 => 69%
844 B654321/S40 => 69%
B731/S765420 => 69%
846 B7421/S76430 => 69%
B74321/S75 => 69%
848 B7531/S6542 => 69%
B7531/S742 => 69%
850 B75431/S6420 => 69%
B75431/S76420 => 69%
852 B754321/S540 => 69%
B754321/S650 => 69%
854 B754321/S6540 => 69%
B754321/S7540 => 69%
856 B754321/S7650 => 69%
B754321/S76540 => 69%
858 B76421/S6430 => 69%
B76421/S643 => 69%
860 B76421/S76430 => 69%
B76421/S7643 => 69%
862 B31/S765420 => 70%
B4321/S750 => 70%
864 B531/S6542 => 70%
B531/S7642 => 70%
866 B531/S76542 => 70%
B54321/S0 => 70%
868 B631/S420 => 70%

B6421/S643 => 70%
870 B64321/S65 => 70%
B64321/S7540 => 70%
872 B64321/S765 => 70%
B65431/S520 => 70%
874 B7421/S6430 => 70%
B7421/S643 => 70%
876 B7421/S7643 => 70%
B74321/S765 => 70%
878 B7531/S7642 => 70%
B7531/S76542 => 70%
880 B754321/S40 => 70%
B754321/S70 => 70%
882 B754321/S7640 => 70%
B7631/S420 => 70%
884 B7631/S6420 => 70%
B764321/S50 => 70%
886 B764321/S65 => 70%
B764321/S750 => 70%
888 B764321/S7540 => 70%
B421/S643 => 71%
890 B42/S76531 => 71%
B4321/S65 => 71%
892 B4321/S765 => 71%
B5431/S20 => 71%
894 B54321/S40 => 71%
B54321/S750 => 71%
896 B631/S6420 => 71%
B642/S31 => 71%
898 B642/S731 => 71%
B64321/S540 => 71%
900 B65431/S6520 => 71%
B65431/S7520 => 71%
902 B65431/S76520 => 71%
B654321/S0 => 71%
904 B654321/S60 => 71%
B654321/S750 => 71%
906 B74321/S50 => 71%
B74321/S65 => 71%
908 B74321/S750 => 71%
B75431/S20 => 71%
910 B75431/S720 => 71%
B754321/S0 => 71%
912 B7642/S31 => 71%
B7642/S731 => 71%
914 B764321/S540 => 71%
B31/S420 => 72%
916 B5431/S720 => 72%
B54321/S50 => 72%
918 B54321/S60 => 72%
B54321/S70 => 72%
920 B631/S7420 => 72%
B65431/S620 => 72%
922 B654321/S50 => 72%
B731/S420 => 72%
924 B742/S76531 => 72%
B754321/S750 => 72%
926 B7631/S7420 => 72%
B31/S6420 => 73%
928 B54321/S740 => 73%
B631/S76420 => 73%
930 B6531/S75420 => 73%
B654321/S70 => 73%
932 B654321/S760 => 73%
B731/S6420 => 73%
934 B742/S6531 => 73%
B75431/S7620 => 73%
936 B754321/S50 => 73%
B754321/S60 => 73%
938 B754321/S740 => 73%
B754321/S760 => 73%
940 B7631/S76420 => 73%
B31/S7420 => 74%
942 B42/S6531 => 74%
B42/S7531 => 74%
944 B5431/S7520 => 74%
B5431/S7620 => 74%
946 B5431/S76520 => 74%
B54321/S760 => 74%
948 B6531/S5420 => 74%
B731/S7420 => 74%
950 B742/S7531 => 74%
B75431/S520 => 74%
952 B75431/S6520 => 74%
B75431/S7520 => 74%
954 B75431/S76520 => 74%
B764321/S7650 => 74%
956 B31/S76420 => 75%
B42/S531 => 75%
958 B5431/S520 => 75%

```
B5431/S6520 => 75%
B642/S76531 => 75%
B64321/S650 => 75%
B64321/S7650 => 75%
B731/S76420 => 75%
B74321/S7650 => 75%
B75431/S620 => 75%
B7642/S76531 => 75%
B764321/S650 => 75%
B4321/S650 => 76%
B4321/S7650 => 76%
B5431/S620 => 76%
B642/S7531 => 76%
B742/S531 => 76%
B74321/S650 => 76%
B531/S5420 => 77%
B531/S75420 => 77%
B642/S6531 => 77%
B6531/S65420 => 77%
B6531/S765420 => 77%
B7531/S5420 => 77%
B7531/S75420 => 77%
B7642/S6531 => 77%
B7642/S7531 => 77%
B642/S531 => 78%
B7642/S531 => 78%
B531/S65420 => 81%
B531/S765420 => 81%
B7531/S65420 => 81%
B7531/S765420 => 81%
B6531/S420 => 83%
B6531/S7420 => 84%
B531/S420 => 85%
B7531/S420 => 85%
B531/S7420 => 87%
B6531/S6420 => 87%
B7531/S7420 => 87%
B531/S6420 => 89%
B6531/S76420 => 89%
B7531/S6420 => 89%
B531/S76420 => 91%
B7531/S76420 => 91%
```

7.2 ColourMap

```
import java.math.BigInteger;
import java.util.Iterator;
import java.util.LinkedList;

/**
 *
 * @author talas
 */
public class ColorMap {

    private static final byte MAX_ELIMINATION_SIZE = 2;

    private final byte[] colors;
    public final byte numColors;
    public final LinkedList<Byte>[] aviColors;
    protected byte[] aviColorsCounter;

    public final long startComplexity;

    private boolean broken = false;
    private boolean crashed = false;

    private boolean[] notBorn;
    private boolean[] notSurvived;

    private LinkedList<Elimination> eliminations;

    private LinkedList<Integer>[] colorRelations;

    public void gotoRandom(){
        boolean needRecurse = false;
        {
            byte[] randColors = new byte[colors.length];
            for(int i = 0; i < randColors.length; i++){
                randColors[i] = aviColors[i].get((byte)(Math.random()*aviColors[i].size()));
            }
            if(!this.tryGoto(randColors)){
                needRecurse = true;
```

```
            }
        }
        if(needRecurse){
            gotoRandom();
        }
    }

    private void optimizeEliminations()
    {
        // find duplicate and overlaping eliminations and remove them..
        System.out.println("E:␣"+eliminations.size());
        int timeOut = Math.max(eliminations.size()/10, 6000);
        {
            LinkedList<Elimination> eliminated = new LinkedList<Elimination>();

            for(Elimination e : eliminations){
                if(e.checked)
                    continue;
                for(Elimination s : eliminations){
                    if(e.containedIn(s)){
                        eliminated.add(s);
                        break;
                    }
                }
                e.checked = true;
                if(--timeOut == 0)
                    break;
            }
            for(Elimination i : eliminated)
                eliminations.remove(i);
        }
        if(timeOut == 0){
            System.out.println("restart!");
            optimizeEliminations();
        }
    }

    public boolean tryGoto(byte[] givenColors){
        if(colors.length != givenColors.length)
            return false;
        for(int i = 0; i < givenColors.length; i++){
            colors[i] = givenColors[i];
            aviColorsCounter[i] = 0;
            boolean found = false;
            for(byte j = 0; j < aviColors[i].size(); j++){
                if(aviColors[i].get(j) == givenColors[i])
                {
                    aviColorsCounter[i] = j;
                    found = true;
                    break;
                }
            }
            if(!found)
                return false;
        }
        return true;
    }

    private final class Elimination {

        /**
         * count: how many number/color pairs in this elimination
         */
        final byte count;
        final int[] elimNumbers;
        final byte[] elimColors;
        boolean checked;
        //public boolean discarded = false;

        @Override
        public String toString(){
            StringBuilder sb = new StringBuilder();
            for(int i = 0; i < elimNumbers.length; i++){
                if(i != 0)
                    sb.append("␣");
                sb.append(elimNumbers[i]);
                sb.append(":");
                sb.append(elimColors[i]);
            }
            return sb.toString();
        }

        // Single elimination
        public Elimination(int num, byte color){
            count = 1;
            elimNumbers = new int[]{num};
            elimColors = new byte[]{color};
        }
```

```
130         // Double elimination
            public Elimination(int num1, int num2, byte color1, byte color2){
132             count = 2;

134             elimNumbers = new int[count];
                elimNumbers[0] = num1;
136             elimNumbers[1] = num2;

138             elimColors = new byte[count];
                elimColors[0] = color1;
140             elimColors[1] = color2;
            }
142
            // X elimination
144         public Elimination(int[] numbers, byte[] colors){
                count = (byte)Math.min(numbers.length, colors.length);
146
                if(count < 1)
148             {
                    System.err.println("Bugged Elimination created, count < 1!");
150             }
                this.elimNumbers = numbers;
152
                this.elimColors = colors;
154         }

156         public final boolean isEliminated(){
                if(count == 1) {
158                 if(colors[elimNumbers[0]] != elimColors[0])
                        return false;
160
                    return true;
162             }
                else if(count < 2) {
164                 // 'this' is a bugged elimination..
                    return false;
166             }

168             // the current colors are only eliminated by this elimination if all colors match.
                for(byte i = 0; i < elimNumbers.length; i++){
170                 if(colors[elimNumbers[i]] != elimColors[i]){
                        return false;
172                 }
                }
174             return true;

176         }

178         public final boolean containedIn(Elimination container){

180             // Check if container has atleast all the same num/color combinations..
                for(byte i = 0; i < this.elimNumbers.length; i++){
182                 int num1 = this.elimNumbers[i];
                    byte color1 = this.elimColors[i];
184
                    boolean found = false;
186                 for(byte j = 0; j < container.elimNumbers.length; j++){
                        if(container.elimNumbers[j] == num1){
188                         if(container.elimColors[j] != color1){
                                // Not the same number/color pair
190                             return false;
                            }
192                         found = true;
                            break;
194                     }
                    }
196                 if(!found)// eliminations doesnt have the same numbers..
                        return false;
198             }
                return (container.count > this.count);
200         }

202         public final boolean sameAs(Elimination other){
                // those that have the same colors can be swapped..
204
                for(byte i = 0; i < this.elimNumbers.length; i++){
206                 int num1 = this.elimNumbers[i];
                    byte color1 = this.elimColors[i];
208
                    boolean found = false;
210                 for(byte j = 0; j < other.elimNumbers.length; j++){
                        if(other.elimNumbers[j] == num1){
212                         if(other.elimColors[j] != color1){
                                return false;
214                         }
                            found = true;
216                         break;
                        }
218                 }
                    if(!found)// eliminations doesnt have the same numbers..
```

```
220                         return false;
                }
222             return true;

224

            }
226     }

228     public final void optimize(byte[] ruleB, byte[] ruleS, Map m)
        {
230         for(int i = 0; i < aviColors.length; i++){
                LinkedList<Byte> current = aviColors[i];
232
                if(current.size() == 0){
234                 this.broken = true;
                    return;
236             }
                Iterator<Byte> iterator = current.iterator();
238
                boolean canBorn = false;
240             boolean canSurvive = false;
                while(iterator.hasNext()){
242                 byte avi = iterator.next().byteValue();
                    if(avi >= ruleB.length){
244                     canSurvive = true;
                    }
246                 else
                        canBorn = true;
248             }

250             if(!canBorn){
                    this.notBorn[i] = true;
252             }
                else if(!canSurvive){
254                 this.notSurvived[i] = true;
                }
256
            }
258

260         PopulationIterator pi = new PopulationIterator(m);

262         LinkedList<Integer> NWCornerDwellers = new LinkedList<Integer>();

264         int current = 0;
            while(pi.hasNext()){
266             // for each color, make eliminations based on available and relations
                pi.next();
268             int[] yx = pi.getXY();

270             int x = yx[1];
                int y = yx[0];
272

274             if((x == 0 || x == 1) && (y == 0 || y == 1))
                    NWCornerDwellers.add(current);
276
                int available = Map.getAvailableNeighbours(x, y, m.getMap().length);
278
                LinkedList<Byte> avi = this.aviColors[current];
280             Iterator<Byte> colorIterator = avi.iterator();

282             int numRels = this.colorRelations[current].size();

284             int numBorn = 0;
                int numSurvive = 0;
286             Iterator<Integer> iterator1 = this.colorRelations[current].iterator();
                while(iterator1.hasNext()){
288                 // find out, for each relation if it must be born, survived or either.
                    int i = iterator1.next();
290                 if(this.notBorn[i]){
                        numSurvive++;
292                 }
                    else if(this.notSurvived[i]){
294                     numBorn++;
                    }
296             }

298             boolean[] eliminatedColors = new boolean[18];
                while(colorIterator.hasNext()){
300                 // check if enough available even when considering relations
                    byte color = colorIterator.next();
302
                    int countDoubles = 0;
304
                    byte currentWanted = 0;
306
                    if(color >= ruleB.length){
308                     currentWanted = ruleS[color-ruleB.length];
                    }
```

```
310                 else
                        currentWanted = ruleB[color];
312
                    if(numSurvive > currentWanted){
314                     // impossible
                        // there are too many neighbours that MUST be true for this color.
316                     // eliminate this WHOLE color for this cell..
                        eliminatedColors[color] = true;
318                     continue;
                    }
320                 else if(available-numBorn < currentWanted){
                        // impossible
322                     // there is NOT enough neighbours that can be true for this color.
                        // eliminate this WHOLE color for this cell..
324                     eliminatedColors[color] = true;
                        continue;
326                 }
                    else {
328                     if(available-numRels < currentWanted){
                            // some eliminations could be had?
330                         // because we depend on some rels to satisfy this color.
332
                            int xElim = (available - currentWanted)+1;// how many required to break this
                                color
334
                            if(xElim == 1){
336                             // double elimination
                                for(Integer other : this.colorRelations[current])
338                             {
                                    for(Byte otherColor : this.aviColors[other]){
340                                     //if otherColor is in ruleB, we can eliminate..
                                        if(otherColor < ruleB.length)
342                                     {
                                            //boolean discarded = false;
344                                         if(countDoubles < current){
                                                Iterator<Elimination> iterator =
                                                    this.eliminations.iterator();
346
                                                Elimination n = new
                                                    Elimination(current,other,color,otherColor);
348                                             boolean found = false;
                                                while(iterator.hasNext()){
350                                                 Elimination t = iterator.next();
                                                    if(t.sameAs(n)){
352                                                     found = true;
                                                        break;
354                                                 }
                                                }
356                                             if(!found){
                                                    countDoubles--;
358                                             }
                                                if(found){
360                                                 //discarded = true;
                                                    //countDoubles++;
362                                                 //continue;
                                                }
364                                             // check if it ACTUALLY should be discarded..

366
                                            }
368
370                                         Elimination n = new Elimination(current,other,color,otherColor);
                                            //n.discarded = discarded;
372                                         this.eliminations.add(n);
                                            countDoubles++;
374                                     }
                                    }
376                             }
                            }
378                         else {

380                             //pick x and x colors to eliminate
                                if(xElim > numRels)
382                                 continue; // cant satisfy this x..????

384                             if(xElim > MAX_ELIMINATION_SIZE)
                                    continue;
386
                                Combinations c = new Combinations(numRels, xElim);
388

390                             while(c.hasNext()){
                                    int[] elim = c.next();
392
                                    byte[] mins = new byte[elim.length];
394                                 byte[] maxs = new byte[elim.length];

396                                 for(int i = 0; i < elim.length; i++){
```

```
                                mins[i] = 99;
398                             maxs[i] = -99;
                                //System.out.append(" "+this.colorRelations[current].get(elim[i]));
400                         }
                            //System.out.println("<");
402                         for(int i = 0; i < elim.length; i++){

404                             // for each i in elim, also permute its colors that can kill..
                                int other = this.colorRelations[current].get(elim[i]);
406                             for(byte j = 0; j < this.aviColors[other].size(); j++){
                                    Byte aviColor = this.aviColors[other].get(j);
408                                 if(aviColor < ruleB.length){
                                        if(j < mins[i])
410                                         mins[i] = j;
                                        if(j > maxs[i])
412                                         maxs[i] = j;
                                    }
414                             }
                            }
416
                            Permutations p = new Permutations(mins, maxs);
418
                            byte[] pElim = null;
420                         do {
                                pElim = p.next();
422                             if(pElim != null){
                                    int[] numbers = new int[pElim.length+1];
424                                 numbers[0] = current;
                                    byte[] currentColors = new byte[pElim.length+1];
426                                 currentColors[0] = color;

428                                 for(int i = 0; i < pElim.length; i++){
                                        int other = this.colorRelations[current].get(elim[i]);
430                                     numbers[i+1] = other;
                                        currentColors[i+1] = this.aviColors[other].get(pElim[i]);
432                                 }

434

436                                 Elimination n = new Elimination(numbers, currentColors);
                                    //System.out.println("E+ "+n);
438                                 this.eliminations.add(n);

440

                                }
442
                            }while(pElim != null);
444
                        }
446
                    }
448             }
                else if(numRels > currentWanted){
450                 // in some cases we could get too many, so can eliminate some here too..

452                 int xElim = currentWanted+1;// how many required to break this color

454                 if(xElim == 1){
                        // double elimination
456                     for(Integer other : this.colorRelations[current])
                        {
458                         for(Byte otherColor : this.aviColors[other]){
                                //if otherColor is in ruleS, we can eliminate..
460                             if(otherColor >= ruleB.length)
                                {
462                                 Elimination n = new Elimination(current,other,color,otherColor);
                                    //System.out.println("E- "+n);
464                                 this.eliminations.add(n);

466                             }
                            }
468                     }
                    }
470                 else {

472                     if(xElim > numRels)
                            continue; // cant satisfy this x..????
474
                        if(xElim > MAX_ELIMINATION_SIZE)
476                         continue;

478                     Combinations c = new Combinations(numRels, xElim);

480                     while(c.hasNext()){
                            int[] elim = c.next();
482
                            byte[] mins = new byte[elim.length];
484                         byte[] maxs = new byte[elim.length];

486                         for(int i = 0; i < elim.length; i++){
```

```
                                    mins[i] = 99;
488                                 maxs[i] = -99;
                                    //System.out.append(" "+this.colorRelations[current].get(elim[i]));
490                             }
                                //System.out.println("<");
492                             for(int i = 0; i < elim.length; i++){

494                                 // for each i in elim, also permute its colors that can kill..
                                    int other = this.colorRelations[current].get(elim[i]);
496                                 for(byte j = 0; j < this.aviColors[other].size(); j++){
                                        Byte aviColor = this.aviColors[other].get(j);
498                                     if(aviColor >= ruleB.length){ // is inside ruleS!
                                            if(j < mins[i]){
500                                             mins[i] = j;
                                            }
502                                         if(j > maxs[i]){
                                                maxs[i] = j;
504                                         }
                                        }
506                                 }
                                }
508
                                Permutations p = new Permutations(mins, maxs);
510
                                byte[] pElim = null;
512                             do {
                                    pElim = p.next();
514                                 if(pElim != null){
                                        int[] numbers = new int[pElim.length+1];
516                                     numbers[0] = current;
                                        byte[] currentColors = new byte[pElim.length+1];
518                                     currentColors[0] = color;

520                                     for(int i = 0; i < pElim.length; i++){
                                            int other = this.colorRelations[current].get(elim[i]);
522                                         numbers[i+1] = other;
                                            currentColors[i+1] = this.aviColors[other].get(pElim[i]);
524                                     }

526                                     Elimination n = new Elimination(numbers, currentColors);
                                        //System.out.println("E+ "+n);
528                                     this.eliminations.add(n);

530                                 }

532                             }while(pElim != null);

534                         }
                        }
536                 }
            }
538         }
            colorIterator = avi.iterator();
540         for(int i = 0; i < eliminatedColors.length; i++){
                // do the actual elimination..
542             if(eliminatedColors[i] == true){
                    //avi.removeFirstOccurrence(i); not good, we depend on the index later?
544                 Elimination n = new Elimination(current,avi.get(i));
                    this.eliminations.add(n);
546                 System.out.println(current+"--"+i);
                }
548         }

550         // finally..
            current++;
552     }

554
        // TODO: Greetings gentlemen, this is corner control.
556     // go thru each corner and check its color stuffs.
        // eliminate combinations that create 'prisoners with no hope'.
558
        // start with northwest corner..
560     // check if it has atleast 2 dots adjacent to it, otherwise no elimination??
        /*{// North-West
562         if(NWCornerDwellers.size() == 3){
                // must be room for some eliminations here..
564             // if all of them are born AND atleast one requires the corner -> corner will live and
                    have zero
                // if all of them survive AND atleast one requires the corner -> corner will live and
                    have 3
566             // if all of them survive AND none of them requires the corner -> corner is dead and
                    have 3
                boolean allBorn = true;
568             boolean allSurvive = true;
                for(Integer cur : NWCornerDwellers){
570                 if(this.notBorn[cur])
                        allBorn = false;
572                 if(this.notSurvived[cur])
                        allSurvive = false;
```

```
574                 }
                    if(allBorn){
576                     // check if atleast 1 requires the corner..
                        //boolean reqFound = false;
578                     /*for(Integer cur : NWCornerDwellers){

580                     }*/
                    /*}
582                 if(allSurvive){
                        // check if atleast 1 requires the corner..
584
                    }
586             }
            }*/
588         System.out.println("Optimizing..");
            this.optimizeEliminations();
590         System.out.println("Done optimizing eliminations.");
        }
592
        public ColorMap(ColorMap copyMap){
594         this.aviColors = copyMap.aviColors;
            this.broken = copyMap.broken;
596         this.eliminations = copyMap.eliminations;
            this.numColors = copyMap.numColors;
598         this.startComplexity = copyMap.startComplexity;
            this.aviColorsCounter = new byte[copyMap.aviColorsCounter.length];
600         this.colors = new byte[copyMap.colors.length];

602         for(int i = 0; i < colors.length; i++)
                colors[i] = aviColors[i].getFirst().byteValue();
604     }


606
        /**
608      * "Blank" ColorMap
         * max size <= Integer.MAX_VALUE
610      * @param size
         */
612     @SuppressWarnings("unchecked")
        public ColorMap(int size, byte[] ruleB, byte[] ruleS, Map map){
614         this.colors = new byte[size];
            this.numColors = (byte) ((byte)ruleB.length+ruleS.length);
616         this.aviColors = new LinkedList[size];
            this.aviColorsCounter = new byte[size];
618         this.eliminations = new LinkedList<Elimination>();
            this.colorRelations = new LinkedList[size];
620
            this.notBorn = new boolean[size];
622         this.notSurvived = new boolean[size];

624         for(int i = 0; i < aviColors.length; i++)
                aviColors[i] = new LinkedList<Byte>();
626
            PopulationIterator pi = new PopulationIterator(map);
628
            Position[] positions = new Position[size];
630
            int currentCellNumber = 0;
632         StringBuilder complexityString = new StringBuilder();
            complexityString.append("CM Perms: ");
634         //complexityString.append("Complexity(?): ");
            long complexity = 1;
636

638         {   //Initial cycle to find all the positions..
                int ccl = 0;
640             PopulationIterator pi_tmp = new PopulationIterator(map);
                while(pi_tmp.hasNext()){
642                 pi_tmp.next();
                    int[] yx = pi_tmp.getXY();
644
                    positions[ccl++] = new Position(yx[1],yx[0]);
646             }
            }
648
            while(pi.hasNext()){
650             pi.next();
                int[] yx = pi.getXY();
652


654
                int x = yx[1];
656             int y = yx[0];

658             int available = Map.getAvailableNeighbours(x, y, map.getMap().length);
                this.colorRelations[currentCellNumber] = new LinkedList<Integer>();
660             for(int p = 0; p < positions.length; p++){
                    if(positions[p].isNextTo(positions[currentCellNumber])){
662                     this.colorRelations[currentCellNumber].add(p);
                    }
```

```
664             }
666
                for(byte i = 0; i < ruleB.length; i++){
668                 if(ruleB[i] <= available)
                        aviColors[currentCellNumber].add((Byte)i);
670             }
672             byte l = (byte)ruleB.length;
                for(byte i = 0; i < ruleS.length; i++){
674                 if(ruleS[i] <= available){
                        byte n = (byte) (i+l);
676                     aviColors[currentCellNumber].add(n);
                    }
678             }
680             if(aviColors[currentCellNumber].size() <= 0){
                    broken = true;
682                 startComplexity = 0;
                    return;
684             }
686             System.out.append(currentCellNumber+":");
                for(int i = 0; i < aviColors[currentCellNumber].size();i++){
688                 System.out.append("␣"+aviColors[currentCellNumber].get(i));
                }
690             System.out.println();
692             if(currentCellNumber != 0)
                    complexityString.append("␣*␣");
694
                complexity *= aviColors[currentCellNumber].size();
696             complexityString.append(aviColors[currentCellNumber].size());
698             colors[currentCellNumber] = aviColors[currentCellNumber].getFirst().byteValue();
                currentCellNumber++;
700         }
            complexityString.append("␣=␣");
702         complexityString.append(complexity);
            startComplexity = complexity;
704
706         this.optimize(ruleB, ruleS, map);
708
            StringBuilder cp2 = new StringBuilder();
710         cp2.append("Neo_Complexity:␣");
            BigInteger n = BigInteger.ONE;
712         for(int i = 0; i < this.aviColors.length; i++){
                n = n.multiply(BigInteger.valueOf(this.aviColors[i].size()));
714         }
716         // add eliminations..
            for(Elimination e : this.eliminations){
718
                long tmp = 1;
720             int c = 0;
                for(int i = 0; i < this.aviColors.length; i++){
722
                    boolean found = false;
724                 for(int j = 0; j < e.elimNumbers.length; j++){
                        if(e.elimNumbers[j] == i)
726                         found = true;
                    }
728                 if(!found){
                        tmp *= this.aviColors[i].size();
730                     c++;
                    }
732             }
734
            }
736
            cp2.append("␣=␣");
738         cp2.append(n);
740
            System.out.println(cp2);
742
            System.out.println("Decimal␣places:␣" + (n.toString().length()-1));
744
    }
746
    public boolean isBroken(){
748     return broken;
    }
750
    public void print(){
752     for(int i = 0; i < colors.length; i++) {
            System.out.append(""+colors[i]);
```

```
754             }
                System.out.println(".");
756         }

758         public int size() {
                return colors.length;
760         }

762         public byte getColor(int pos){ // throws ArrayOutOfBoundsException
                return colors[pos];
764         }

766         public Byte[] getColors(){
                Byte[] b = new Byte[colors.length];
768             int i = 0;
                for(byte c : colors)
770                 b[i++]=c;
                return b;
772         }

774         public void gotoNext(){
                if(numColors <= 1)
776                 return; // Nothing to do.



780             if(!hasNext())
                    return;
782



784             add(colors.length-1);
786             //this.gotoRandom(); an option to iterating..

788             crashed = false;
                for(Elimination e : eliminations){
790                 if(e.isEliminated()){
                        try{
792                         do {
                                add(colors.length-1);
794                         } while(e.isEliminated());
                        }
796                     catch(java.lang.StackOverflowError soe){
                            crashed = true;
798                         //System.out.println("*");
                            break;
800                     }
                        break;
802                 }
                }
804         }

806
            public boolean hasCrashed(){
808             return crashed;
            }
810
            public boolean hasNext(){
812             if(numColors <= 1)
                    return false;
814
                for(int i = 0; i < colors.length; i++){
816                 if(colors[i] != aviColors[i].getLast().byteValue()){
                        if(aviColorsCounter[i] != aviColors[i].size()-1)
818                         return true;
                    }
820             }
                return false;
822         }

824         private void add(int pos){
                if(pos < 0 || pos >= colors.length){
826                 this.broken = true;
                    return;
828             }

830             if(aviColorsCounter[pos] < aviColors[pos].size()-1){
                    colors[pos] = aviColors[pos].get(++aviColorsCounter[pos]);
832                 return;
                }
834             else {
                    aviColorsCounter[pos] = 0;
836                 if(aviColors[pos].size() == 0)
                        return;
838                 colors[pos] = aviColors[pos].getFirst().byteValue();
                    add(pos-1);
840             }
            }
842 }
```

7.3 PermMap

```
2  import java.util.Iterator;
   import java.util.LinkedList;
4
   /**
6   *
    * @author talas
8   */
   public final class PermMap implements Runnable{
10     private final byte[][] map;

12     private final byte[][] neighbourMap;

14     private final boolean[][] doneMap;

16     private final boolean[][] dontCareMap;

18     public static final byte FINAL_DEAD = 3;
       public static final byte FINAL_ALIVE = 4;
20     public static final byte DIES_NOW = 5;
       public static final byte DUNNO = 0;
22     public static final byte CAN = 1;
       public static final byte BORN = 2;
24
       private static final byte[] ruleB = Global.ruleB;
26     private static final byte[] ruleS = Global.ruleS;

28     private boolean hasSolution = false;
       private boolean hasForks = false;
30
       private static long superIt = 0;
32     private long myIt = superIt++;

34     public void setStaticMap(boolean[][] smap){
           for(byte i = 0; i < smap.length; i++){
36             for(byte j = 0; j < smap.length; j++){
                   if(smap[i][j] == true){
38                     if(map[i][j] != FINAL_ALIVE){
                           map[i][j] = DIES_NOW;
40                     }
                   }
42             }
           }
44     }


46
       public static final boolean debug = false;
48
       //public Position[] failReason;
50
       private Map solution;
52
       public long getScore(){
54         long n = 0;
           for(int i = 0; i < doneMap.length; i++){
56             for(int j = 0; j < doneMap.length; j++){
                   if(doneMap[i][j]){
58
                       byte code = map[i][j];
60                     if(code == FINAL_ALIVE || code == BORN)
                           n += 10;
62                     else
                           n += 1;
64                 }

66             }
           }
68


70         return n;
       }
72
       public byte[][] getTempMap(){
74         return map;
       }
76
       public byte[][] getNeighbourMap(){
78         return neighbourMap;
       }
80
       public Map buildSolution(){
82         if(!hasSolution)
               return null;
84         if(solution != null)
               return solution;
86         Map m = new Map(map.length);
```

```
88          for(int i = 0; i < map.length; i++){
                for(int j = 0; j < map.length; j++){
90                  m.setState(i, j, (map[i][j]== FINAL_ALIVE || map[i][j] == DIES_NOW));
                }
92          }
            solution = m.clone();
94          return m;
        }
96
        public boolean hasForks() { return hasForks; }
98

100     private PermMap(byte[][] map, byte[][] neighbourMap, boolean[][] doneMap){
            this.map = map;
102         this.neighbourMap = neighbourMap;
            this.doneMap = doneMap;
104         this.dontCareMap = null;
            if(debug){
106             System.out.println("a+");
                Global.printMap(map);
108             Global.printMap(neighbourMap);
                Global.printMap(doneMap, 'd', 'n');
110             System.out.println("a-");
            }
112     }

114     private PermMap(byte[][] map, byte[][] neighbourMap, boolean[][] doneMap, boolean[][] dontCareMap){
            this.map = map;
116         this.neighbourMap = neighbourMap;
            this.doneMap = doneMap;
118         this.dontCareMap = dontCareMap;

120         if(debug){
                System.out.println("a+");
122             Global.printMap(map);
                Global.printMap(neighbourMap);
124             Global.printMap(doneMap, 'd', 'n');
                System.out.println("a-");
126         }
        }
128
        public PermMap(Byte[] def, Position[] dots, byte size){
130
            map = new byte[size][size];
132         neighbourMap = new byte[size][size];
            doneMap = new boolean[size][size];
134         this.dontCareMap = null;
            for(int i = 0; i < neighbourMap.length; i++){
136             for(int j = 0; j < neighbourMap.length; j++){
                    neighbourMap[i][j] = -1;
138             }
            }
140
            for(int i = 0; i < dots.length; i++){
142             Position current = dots[i];
                byte color = def[i];
144
                boolean born = true;
146             if(color >= ruleB.length){
                    born = false;
148                 neighbourMap[current.x][current.y] = ruleS[color-ruleB.length];
                }
150             else
                    neighbourMap[current.x][current.y] = ruleB[color];
152
                if(born == true) {
154                 map[current.x][current.y] = BORN;
                }
156             else
                    map[current.x][current.y] = FINAL_ALIVE;
158             Global.ins++;
            }
160     }

162     public PermMap(Byte[] def, Position[] dots, byte size, boolean[][] dontCareMap){

164         map = new byte[size][size];
            neighbourMap = new byte[size][size];
166         doneMap = new boolean[size][size];
            this.dontCareMap = dontCareMap;
168
            for(byte i = 0; i < neighbourMap.length; i++){
170             for(byte j = 0; j < neighbourMap.length; j++){
                    neighbourMap[i][j] = -1;
172             }
            }
174
            for(int i = 0; i < dots.length; i++){
176             Position current = dots[i];
```

```
            byte color = def[i];

            boolean born = true;
            if(color >= ruleB.length){
                born = false;
                neighbourMap[current.x][current.y] = ruleS[color-ruleB.length];
            }
            else
                neighbourMap[current.x][current.y] = ruleB[color];

            if(born == true) {
                map[current.x][current.y] = BORN;
            }
            else
                map[current.x][current.y] = FINAL_ALIVE;
            Global.ins++;
        }
    }

    public PermMap(ColorMap def, Position[] dots, byte size){

        map = new byte[size][size];
        neighbourMap = new byte[size][size];
        doneMap = new boolean[size][size];
        this.dontCareMap = null;
        for(int i = 0; i < neighbourMap.length; i++){
            for(int j = 0; j < neighbourMap.length; j++){
                neighbourMap[i][j] = -1;
            }
        }

        for(int i = 0; i < dots.length; i++){
            Position current = dots[i];
            byte color = def.getColor(i);

            boolean born = true;
            if(color >= ruleB.length){
                born = false;
                neighbourMap[current.x][current.y] = ruleS[color-ruleB.length];
            }
            else
                neighbourMap[current.x][current.y] = ruleB[color];

            if(born == true) {
                map[current.x][current.y] = BORN;
            }
            else
                map[current.x][current.y] = FINAL_ALIVE;
            Global.ins++;
        }
    }

    public PermMap(ColorMap definition, Map realMap){
        byte[] perms = new byte[definition.size()];
        map = new byte[realMap.getMap().length][realMap.getMap().length];
        neighbourMap = new byte[realMap.getMap().length][realMap.getMap().length];
        doneMap = new boolean[realMap.getMap().length][realMap.getMap().length];
        this.dontCareMap = null;
        for(int i = 0; i < neighbourMap.length; i++){
            for(int j = 0; j < neighbourMap.length; j++){
                neighbourMap[i][j] = -1;
            }
        }
        byte[] neighbours = new byte[definition.size()];

        for(int i = 0; i < definition.size(); i++){

            byte color = definition.getColor(i);
            boolean born = true;
            if(color >= ruleB.length){
                born = false;
                neighbours[i] = ruleS[color-ruleB.length];
            }
            else
                neighbours[i] = ruleB[color];

            if(born == true) {
                perms[i] = BORN;
            }
            else
                perms[i] = FINAL_ALIVE;
        }

        PopulationIterator pi = new PopulationIterator(realMap);
        while(pi.hasNext()){
            pi.next();
            int[] xy = pi.getXY();

            if(xy != null){
                //System.out.println("x:"+xy[0] + ", y"+xy[1]+", c:"+(++count));
```

```
                map[xy[1]][xy[0]] = perms[(int)pi.getCurrent()];
268             Global.ins++;
                neighbourMap[xy[1]][xy[0]] = neighbours[(int)pi.getCurrent()];
270         }
        }
272     if(debug){
            System.out.println("b+");
274         Global.printMap(map);
            Global.printMap(neighbourMap);
276         System.out.println("b-");
        }
278
    }
280
    public boolean solveDunno(int x, int y){
282     for(int i = -1; i <= 1; i++){
            for(int j = -1; j <= 1; j++){
284             if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length || (i == 0 && j == 0)){
                    continue;
286             }
                byte nWants = neighbourMap[x+i][y+j];
288
                if(nWants != -1){
290                 map[x][y] = CAN;
                    Global.ins++;
292                 neighbourMap[x][y] = -1;
                    return true;
294             }
            }
296     }
        return false;
298 }

300 public byte countNeighbours(int x, int y){
        byte count = 0;
302     for(int i = -1; i <= 1; i++){
            for(int j = -1; j <= 1; j++){
304             if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length || (i == 0 && j == 0)){
                    continue;
306             }

308             byte code = map[x+i][y+j];
                if(code == FINAL_ALIVE || code == DIES_NOW ){
310                 count++;
                }
312         }
        }
314     return count;
    }
316
    public void killCAN(int x, int y){
318     map[x][y] = FINAL_DEAD;
        Global.ins++;
320     keepDead(x,y);
    }
322
    public boolean keepDead(int x, int y){
324     if(this.dontCareMap != null){
            if(this.dontCareMap[x][y]){
326             doneMap[x][y] = true;
                return true; // dont fix 'dont cares'
328         }
        }
330
        boolean dies = (map[x][y] == DIES_NOW);
332     byte countNeighbours = 0;

334     byte canDunnoCount = 0;
        LinkedList<Position> cans = new LinkedList<Position>();
336
        for(byte i = -1; i <= 1; i++){ for(byte j = -1; j <= 1; j++) {
338         int xi = x+i;
            int yj = y+j;
340
            if(xi < 0 || xi >= map.length || yj < 0 || yj >= map.length || (i == 0 && j == 0))
342             continue;

344             byte code = map[xi][yj];

346             if(code == FINAL_ALIVE || code == DIES_NOW)
                    countNeighbours++;
348
                if(code == CAN || code == DUNNO){
350                 canDunnoCount++;
                    cans.add(new Position(xi,yj));
352             }

354
        } }// End for loops
356
```

```
        boolean fail = false;
358     if(!dies) {// !dies = trying to stay dead
            for(byte i = 0; i < ruleB.length; i++){
360             if(ruleB[i] == countNeighbours){
                    fail = true;
362             }
            }
364     }
        else {
366         for(byte i = 0; i < ruleS.length; i++){
                if(ruleS[i] == countNeighbours){
368                 fail = true;
                }
370         }
        }
372
        if(!fail){
374         // solved for the moment..
            neighbourMap[x][y] = countNeighbours;
376         return true;
        }
378

380     byte first = -1;
        for (byte i = countNeighbours; i < 9; i++) {
382         boolean has = false;
            for (byte j = 0; j < (!dies ? ruleB.length : ruleS.length); j++) {
384             if (!dies && ruleB[j] == i) {
                    has = true;
386             } else if (dies && ruleS[j] == i) {
                    has = true;
388             }
            }
390         if (!has) {
                first = i;
392             break; // no need to continue..
            }
394     }

396     if(first == -1) {
            // Cant make this one stay dead...
398         return false;
        }
400
        if (first == countNeighbours + canDunnoCount) {
402         // Only one way to do it.. so.. do it..
            Iterator<Position> iterator = cans.iterator();
404         while(iterator.hasNext()){
                Position p = iterator.next();
406             map[p.x][p.y] = DIES_NOW;
                Global.ins++;
408         }
            neighbourMap[x][y] = -1;
410         doneMap[x][y] = true;
        } else {
412         // set how many we think we need...
            neighbourMap[x][y] = first;
414     }
        return true;
416 }

418 public byte runValue = -99;

420 @Override
    public void run(){
422     runValue = go();
    }
424
    private byte go(){
426
        boolean allDone = false;
428
        int timeOut = 1000;
430
        while(!allDone) {
432
            allDone = true;
434         timeOut--;

436         if(timeOut == 0)
                break;
438
            boolean noChange = true;
440
            for(int x = 0; x < map.length; x++){
442             for(int y = 0; y < map.length; y++){

444
                    if(neighbourMap[x][y] != -1){
446
```

```
                    if(this.countNeighbours(x, y) == neighbourMap[x][y]){
448
                        if(this.map[x][y] == BORN || this.map[x][y] == FINAL_ALIVE)
450                     {
                            // kill off the remaining CANs and DUNNOs..
452                         for(int i = -1; i <= 1; i++){
                            for(int j = -1; j <= 1; j++){
454                             if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length ||
                                      (i == 0 && j == 0)){
                                    continue;
456                             }

458                             byte code = map[x+i][y+j];
                                if(code == CAN || code == DUNNO ){
460                                 killCAN(x+i,y+j);
                                }
462                         } }
                            noChange = false;
464                     }
                        doneMap[x][y] = true;
466                 }
                    else
468                     doneMap[x][y] = false;
                }
470

472             if(doneMap[x][y])
                    continue;
474


476
                byte reqNeighbours = neighbourMap[x][y];
478
                byte availableNeighbours = 8;
480             byte relNeighbours = reqNeighbours;

482



484
                if(map[x][y] == CAN)
486                 continue; // Nothing to do..
                boolean iDunno = (map[x][y] == DUNNO);
488
                if(iDunno) {
490                 boolean tmp = solveDunno(x,y);
                    if(allDone)
492                     allDone = tmp;
                    continue;
494             }

496             allDone = false;

498             availableNeighbours = Map.getAvailableNeighbours(x,y,map.length);

500
                //System.out.println("req: "+reqNeighbours+" / "+availableNeighbours);
502             if(reqNeighbours > availableNeighbours)
                {
504                 return 0; // ColorMap is wrong.. corner case
                }
506
                // 1 check for 'zeros' and nullify neighbours, if !possible -> discard colormap
508             // 2 check for cells that have too many neighbours, if found -> discard colormap
                // 3 check for cells that have been satisfied and nullify neighbours, if !possible
                     -> discard colormap
510             // 4 check if cell has same amount of avi and req. if so, solve req..

512             if(map[x][y] == FINAL_DEAD || map[x][y] == DIES_NOW){
                    boolean test = this.keepDead(x, y);
514                 if(!test)
                        return 7;
516                 noChange = false;
                    continue; // done for now..
518             }

520             byte canCount = 0;

522             for(int i = -1; i <= 1; i++)
                {
524                 for(int j = -1; j <= 1; j++){
                        if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length || (i == 0
                             && j == 0)){
526                         continue;
                        }
528
                        byte code = map[x+i][y+j];
530                     if(code == FINAL_ALIVE || code == DIES_NOW ){
                            if(relNeighbours <= 0)
532                             return 1; // ColorMap is wrong.. too many neighbours for this cell..
```

```
534                                 relNeighbours--;
536                             }
538
                                if(code == CAN || code == DUNNO){
540                                 if(relNeighbours == 0)// set it to dead..
                                    {
542                                     map[x+i][y+j] = FINAL_DEAD;// zero case.. nullify neighbours
                                        Global.ins++;
544                                     noChange = false;
                                    }
546
                                    else
548                                     canCount++;
                                }
550                         }
                        }
552                     if(relNeighbours == 0)
                            doneMap[x][y] = true;
554                     else {
556                         if(canCount < relNeighbours){
                                return 2;
558                         }
                            else if(canCount == relNeighbours){
560                             // Solve
                                for(int i = -1; i <= 1; i++) {
562                                 for(int j = -1; j <= 1; j++) {
                                        if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length ||
                                                (i == 0 && j == 0)){
564                                         continue;
                                        }
566                                     if(map[x+i][y+j] == CAN || map[x+i][y+j] == DUNNO){
                                            map[x+i][y+j] = DIES_NOW;
568                                         Global.ins++;
                                        }
570                                 }
                                }
572                             doneMap[x][y] = true;
                                noChange = false;
574                         }
                        }
576                 } // end for y
                }// end for x
578
                if(noChange)
580                 break; // if nothing changes we're done for now..
582         }// end while
584
            if(debug){
586             System.out.println("f+");
                Global.printMap(map);
588             Global.printMap(neighbourMap);
                Global.printMap(doneMap, 'd', 'n');
590             System.out.println("f-");
            }
592
            boolean solved = true;
594
            for(int i = 0; i < map.length; i++){
596             for(int j = 0; j < map.length; j++){
                    if(map[i][j] != CAN && map[i][j] != DUNNO){
598                     if(!doneMap[i][j])
                            solved = false;
600                 }
                }
602         }

604         if(solved == true){
                hasSolution = true;
606             return -1;
                //System.out.println("PermMap solved!");
608         }
            else {
610             hasForks = true;

612             //System.out.println("/////////// FORKING //////////////");

614             int canCount = 0;
                for(int i = 0; i < map.length; i++){
616                 for(int j = 0; j < map.length; j++){
                        if(map[i][j]== CAN)
618                         canCount++;

620                 }
                }
622
```

```
624             // look for 1/2's.. "either/or's"
                // that is: dots that require 1 and have 2 cans near it.
626             // else: "smallest choice"
                double miniPoints = 0;
628             byte smallestDiv = 99;
                LinkedList<Position> bestChoices = null;
630

632             if(true){
                    for(int x = 0; x < map.length; x++){
634                     for(int y = 0; y < map.length; y++){
                            if(map[x][y] != CAN && map[x][y] != DUNNO){
636                             // check if we want more..
                                byte current = 0;
638                             LinkedList<Position> thisOnesChoices = new LinkedList<Position>();
                                for(int i = -1; i <= 1; i++) {
640                                 for(int j = -1; j <= 1; j++) {
                                        if(x+i < 0 || x+i >= map.length || y+j < 0 || y+j >= map.length ||
                                                (i == 0 && j == 0)){
642                                         continue;
                                        }
644

                                        byte code = map[x+i][y+j];
646

                                        if(code == CAN || code == DUNNO){
648                                         thisOnesChoices.add(new Position(x+i,y+j));
                                        }
650                                     else if(code != BORN && code != FINAL_DEAD){
                                            current++;
652                                     }
                                    }
654                             }
                                byte req = (byte)(this.neighbourMap[x][y]-current);
656                             if(req >= 1 && req <= smallestDiv){

658                                 boolean reallySmaller = (req < smallestDiv);

660                                 double myMinipoints = Math.abs(thisOnesChoices.size()-(req/2.0));

662                                 if(thisOnesChoices.size() >= 1 && (myMinipoints > miniPoints ||
                                            reallySmaller)){
                                        smallestDiv = req;
664                                     miniPoints = myMinipoints;
                                        bestChoices = thisOnesChoices;
666                                     //System.out.println("n:"+myIt);
                                    }
668

                                }
670                         }
                        }
672             }
                }
674

                if(bestChoices != null){
676                 //System.out.println("bestChoices.size(): "+bestChoices.size());

678                 for(int current = 0; current < bestChoices.size(); current++) {

680                     //set number 'current' as DIES_NOW and 'go' at it.

682                     byte[][] copyMap = Global.cloneMap(this.map);
                        byte[][] copyNeighbour = Global.cloneMap(this.neighbourMap);
684                     boolean[][] copyDone = Global.cloneMap(doneMap);

686                     Position currentPos = bestChoices.get(current);

688                     copyMap[currentPos.x][currentPos.y] = DIES_NOW;
                        Global.ins++;
690

                        PermMap newMap = null;
692

                        if(this.dontCareMap == null)
694                         newMap = new PermMap(copyMap, copyNeighbour, copyDone);
                        else
696                         newMap = new PermMap(copyMap, copyNeighbour, copyDone, dontCareMap);

698                     byte value = newMap.go();
                        if(debug){
700                         System.out.println("->>");
                            System.out.println("r"+value);
702                     }
                        if(value == -1){
704                         // check if found..
                            if(newMap.hasSolution){
706                             // ok, set solution.. and return

708                             if(this.hasSolution){
                                        System.out.println("new??");
710                                     solution.printMap();
```

```
                                synchronized(Global.solutions){
712                                 Global.solutions.add(solution);
                                }
714                             //System.out.println("go back.");
                            }
716
                            this.solution = newMap.buildSolution();
718


720
                            this.hasSolution = true;
722                         //return -1;
                        }
724                 }
                }
726             if(this.hasSolution)
                    return -1;
728         }
            /*else if(canCount == 1) { // Older code, brute force to insert dots.
730
                for(int current = 1; current <= canCount; current++) {
732
                    //set number 'current' as DIES_NOW and 'go' at it.
734
                    byte[][] copyMap = Global.cloneMap(this.map);
736                 byte[][] copyNeighbour = Global.cloneMap(this.neighbourMap);
                    boolean[][] copyDone = Global.cloneMap(doneMap);
738
                    int count = 0;
740                 boolean found = false;
                    for(int i = 0; i < copyMap.length; i++){
742                     for(int j = 0; j < copyMap.length; j++){
                            if(copyMap[i][j] == CAN){
744                             count++;

746                             if(count == current){
                                    copyMap[i][j] = DIES_NOW;
748                                 found = true;
                                }
750                         }
                            if(found)
752                             break;
                        }
754                     if(found)
                            break;
756                 }

758                 PermMap newMap = new PermMap(copyMap, copyNeighbour, copyDone);

760                 byte value = newMap.go();
                    if(debug){
762                     System.out.println("<--");
                        System.out.println("r"+value);
764                 }
                    if(value == -1){
766                     // check if found..
                        if(newMap.hasSolution){
768


770
                            // ok, set solution.. and return
772                         this.solution = newMap.buildSolution();
                            this.hasSolution = true;
774                         if(debug){
                                solution.printMap();
776                             System.out.println("go back.");
                            }
778                         return -1;
                        }
780                 }
                }
782         }*/

784
            // Forks.. need to permute..
786     }

788     return -2;
    }
790 }
```

7.4 Map

```
2 import java.io.BufferedOutputStream;
  import java.io.File;
4 import java.io.FileInputStream;
```

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Iterator;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.zip.*;

/**
 *
 * @author talas
 */
public final class Map implements Iterable<Boolean> {

    private int population = 0;

    public void printMap(){

        if(real_map.length > 200){
            System.out.println("Too big, not printing to console.");
            return;
        }

        for(int x = 0; x < real_map.length; x++){
            StringBuilder sb = new StringBuilder();
            for(int y = 0; y < real_map.length; y++){
                if(_getState(x,y))
                    sb.append("x");
                else
                    sb.append("o");
            }
            System.out.println(sb.toString());
        }

    }

    public String getOneliner(){
        StringBuilder sb = new StringBuilder();
        for(int x = 0; x < real_map.length; x++){
            for(int y = 0; y < real_map.length; y++){
                if(_getState(x,y))
                    sb.append("x");
                else
                    sb.append("o");
            }
        }
        return sb.toString();
    }

    public static boolean[][] fromByte(byte[] bytes){
        int size = (int)Math.ceil(Math.sqrt(bytes.length*8));
        boolean[][] map = new boolean[size][size];

        int count = 0;
        for (int i = 0; i < bytes.length; i++) {
            int remainder = bytes[i]+128;
            boolean current = false;

            for(int j = 128; j >= 1; j = (int)Math.floor(j /= (0.0+2))){
                if(remainder >= j) {
                   remainder -= j;
                   current = true;
                }
                else
                    current = false;

                int y = (int)Math.floor(count/map.length+0.0);
                map[count-map.length*y][y] = current;

                count++;
            }

        }
        return map;
    }

    public byte[] toByte(){
        int[] res = new int[(real_map.length*real_map.length)/8];

        int currentbyte = 0;
        int myByte = 0;
        int bitNumber = 0;
        for(int i = 0; i < real_map.length; i++){
            for(int j = 0; j < real_map.length; j++){
                switch(bitNumber){
                    case 0:
                        myByte += ((real_map[j][i]) ? 128 : 0);
                        break;
```

```
                    case 1:
96                      myByte += ((real_map[j][i]) ? 64 : 0);
                        break;
98                  case 2:
                        myByte += ((real_map[j][i]) ? 32 : 0);
100                     break;
                    case 3:
102                     myByte += ((real_map[j][i]) ? 16 : 0);
                        break;
104                 case 4:
                        myByte += ((real_map[j][i]) ? 8 : 0);
106                     break;
                    case 5:
108                     myByte += ((real_map[j][i]) ? 4 : 0);
                        break;
110                 case 6:
                        myByte += ((real_map[j][i]) ? 2 : 0);
112                     break;
                    case 7:
114                     myByte += ((real_map[j][i]) ? 1 : 0);
                        res[currentbyte++] = (int)Math.abs(myByte);
116                     myByte = 0;
                        bitNumber = -1;
118                     break;
                }
120             bitNumber++;
            }
122     }
        byte[] b = new byte[res.length];
124     for(int i = 0; i < res.length; i++) {
            b[i] = (byte)(res[i]-128);
126     }

128     return b;
    }
130
    public int compressMap()
132 {
        try {
134
        int l = 0;
136     int l2 = 0;
        {
138         byte[] sendBuf = toByte();

140         Deflater d = new Deflater();
            d.setInput(sendBuf);
142
            d.finish();
144
            byte[] output = new byte[sendBuf.length];
146
            l = d.deflate(output);
148         int n = (int)d.getBytesWritten();
            n = d.getTotalOut();
150
            d.finish();
152
            n = d.getTotalOut();
154     }

156     {
            byte[] sendBuf = this.RLEncode().getBytes("ASCII");
158
            Deflater d = new Deflater();
160         d.setInput(sendBuf);

162         d.finish();

164         byte[] output = new byte[sendBuf.length];

166         l2 = d.deflate(output);
            int n = (int)d.getBytesWritten();
168         n = d.getTotalOut();

170         d.finish();

172         n = d.getTotalOut();

174     }

176         if(l2 < l)
            {
178             //Choose rle..
                System.out.println("Chose RLE..!");
180             return l2;
            }
182
            return l;
184     } catch (Exception ex) {
```

```
            Logger.getLogger(Map.class.getName()).log(Level.SEVERE, null, ex);
        }
        return Integer.MAX_VALUE;

    }

    public Map getDiff(Map other){
        Map diff = new Map(Map.xorMap(other.getMap(),real_map));
        return diff;
    }

    public static Map xorMap(Map map1, Map map2){
        int mapsize = map1.getMap().length;
        Map res = new Map(mapsize);
        for(int i = 0; i < mapsize; i++) {
            for(int j = 0; j < mapsize; j++) {
                if(map1.getState(i,j) && !map2.getState(i,j)){
                    res.setState(i,j,true);
                }
                if(!map1.getState(i,j) && map2.getState(i,j)){
                    res.setState(i,j,true);
                }
            }
        }
        return res;
    }

    public static boolean[][] xorMap(boolean[][] map1, boolean[][] map2){
        boolean[][] res = new boolean[map1.length][map1.length];
        for(int i = 0; i < map1.length; i++) {
            for(int j = 0; j < map1.length; j++) {
                if(map1[i][j] && !map2[i][j]){
                    res[i][j] = true;
                }
                if(!map1[i][j] && map2[i][j]){
                    res[i][j] = true;
                }
            }
        }
        return res;
    }

    public static byte getAvailableNeighbours(int x, int y, int mapsize) {
        byte availableNeighbours = 8;
        if (x == 0) {
            // Maybe corner
            if (y == 0) {
                return 3;
            } else if (y == mapsize - 1) {
                availableNeighbours -= 5;
            } else {
                availableNeighbours -= 3;
            }
        } else if (x == mapsize - 1) {
            // Maybe corner
            if (y == 0) {
                availableNeighbours -= 5;
            } else if (y == mapsize - 1) {
                availableNeighbours -= 5;
            } else {
                availableNeighbours -= 3;
            }
        } else if (y == 0) {
            availableNeighbours -= 3;
        } else if (y == mapsize - 1) {
            availableNeighbours -= 3;
        }
        return availableNeighbours;
    }

    public static long diff(Map current, Map origin) {
        long numWrong = 0;

        if(current.getMap().length != origin.getMap().length)
            return Integer.MAX_VALUE;

        int size = current.getMap().length;

        for(int i = 0; i < size; i++){
            for(int j = 0; j < size; j++){
                boolean curr = current.getMap()[i][j];
                boolean orig = origin.getMap()[i][j];
                if(curr != orig)
                    numWrong++;
            }
        }

        return numWrong;
    }

```

```
    public static long diff(Map current, Map origin, boolean[][] dontCares) {
276     long numWrong = 0;

278     if(current.getMap().length != origin.getMap().length)
            return Integer.MAX_VALUE;
280
        int size = current.getMap().length;
282
        for(int i = 0; i < size; i++){
284         for(int j = 0; j < size; j++){
                if(dontCares != null && dontCares[i][j])
286                 continue;
                boolean curr = current.getMap()[i][j];
288             boolean orig = origin.getMap()[i][j];
                if(curr != orig)
290                 numWrong++;
            }
292     }

294     return numWrong;
    }
296
    @Override
298 public Map clone()
    {
300     boolean[][] mappy = new boolean[real_map.length][real_map.length];

302     for (int i = 0; i < mappy.length; i++){
            System.arraycopy(real_map[i], 0, mappy[i], 0, mappy.length);
304     }
        return new Map(mappy);
306 }

308 public String RLEncode(){
        // output map as RLE, but only the data, no headers etc. and no $ signs..
310
        StringBuilder code = new StringBuilder();
312
        MapIterator mi = (MapIterator)this.iterator();
314
        boolean last = false;
316     int count = 0;
        int maxCount = 0;
318     boolean first = true;
        while(mi.hasNext()){
320         mi.next();
            if(count == 0){
322             if(!first)
                    code.append(last?"x":"o");
324             else
                    first = false;
326             last = mi.get();
                count++;
328         }
            else if(mi.get() == last){
330             count++;
                continue;
332         }
            else {
334             if(count > 1){
                    if(count > maxCount)
336                     maxCount = count;

338                 code.append(count);
                }
340             code.append(last?"x":"o");
                last = mi.get();
342             count = 1;
            }
344     }

346     if(count > 1){
            if(count > maxCount)
348                 maxCount = count;

350         code.append(count);
        }
352     code.append(last?"x":"o");

354     System.out.println("RLE length: "+code.length());
        System.out.println("RLE max count: "+maxCount);
356
        return code.toString();
358 }

360 public void writeToFile(File f){
        try {
362
            byte[] sendBuf = toByte();
364         FileOutputStream fos = new FileOutputStream(f);
```

```
            BufferedOutputStream bos = new BufferedOutputStream(fos);
366             bos.write(sendBuf, 0, sendBuf.length);
            bos.close();
368             fos.close();
            System.out.println("Map saved: "+sendBuf.length);
370             return;

372         } catch (IOException ex) {
            Logger.getLogger(Map.class.getName()).log(Level.SEVERE, null, ex);
374         }
        System.out.println("FAILed to save map!");
376     }

378     public void writeToCompressedFile(File f){

380         try {

382             byte[] sendBuf = toByte();
            byte[] output = new byte[sendBuf.length];
384
            System.out.println("going to save: " + sendBuf.length + " Bytes..");
386             Deflater d = new Deflater();
            d.setInput(sendBuf);
388
            d.finish();
390             int l = d.deflate(output);
            d.finish();
392             FileOutputStream fos = new FileOutputStream(f);
            BufferedOutputStream bos = new BufferedOutputStream(fos);
394             bos.write(output, 0, l);
            bos.close();
396             fos.close();
            System.out.println("Compressed Map saved: "+l);
398             return;

400         } catch (IOException ex) {
            Logger.getLogger(Map.class.getName()).log(Level.SEVERE, null, ex);
402         }
        System.out.println("Failed to save map!");
404     }

406     public static Map nextGeneration(Map current, byte[] ruleB, byte[] ruleS){
        // Simulate one generation on 'current'
408
        byte mapsize = (byte)current.getMap().length;
410         boolean[][] next = new boolean[mapsize][mapsize];

412         byte[][] neighbourMap = new byte[mapsize][mapsize];

414         for(byte i = 0; i < mapsize; i++){
            for(byte j = 0; j < mapsize; j++){
416                 if(current._getState(i, j)){
                    if(i>0){
418                     if(j>0)
                        neighbourMap[i-1][j-1]++;
420                     neighbourMap[i-1][j]++;
                        if(j<mapsize-1)
422                         neighbourMap[i-1][j+1]++;
                    }
424
                    if(j>0){
426                     neighbourMap[i][j-1]++;
                    }
428
                    if(i < mapsize-1){
430                     if(j>0)
                        neighbourMap[i+1][j-1]++;
432                     neighbourMap[i+1][j]++;
                        if(j<mapsize-1)
434                         neighbourMap[i+1][j+1]++;
                    }
436
                    if(j<mapsize-1)
438                     neighbourMap[i][j+1]++;

440                 }
            }
442         }

444         for(byte i = 0; i < mapsize; i++){
            for(byte j = 0; j < mapsize; j++){
446                 boolean wasAlive = current.getMap()[i][j];
                byte neighbours = neighbourMap[i][j];
448                 if(wasAlive){
                    for(byte r : ruleS){
450                     if(neighbours == r){
                        next[i][j] = true;
452                         break;
                    }
454                 }
```

```
                }
456             else {
                    for(byte r : ruleB){
458                     if(neighbours == r){
                            next[i][j] = true;
460                         break;
                        }
462                 }
                }
464
            }
466     }
        return new Map(next);
468     /* // Older version of code, saved for reference.
        MapIterator it = (MapIterator) current.iterator();
470
        while(it.hasNext()){
472         boolean alive = (Boolean)it._next();
            byte neighbours = it.getNeighbours();
474         if(alive){

476             for(int i = 0; i < ruleS.length; i++){
                    if(neighbours == ruleS[i]){
478                     next.setState(it.getX(), it.getY(), true);
                        break;
480                 }
                }
482
            }
484         else {
                for(int i = 0; i < ruleB.length; i++){
486                 if(neighbours == ruleB[i]){
                        next.setState(it.getX(), it.getY(), true);
488                     break;
                    }
490             }
            }
492     }
        return next;*/
494 }

496 static long compare(Map current, Map origin) {
        // if 100% match then MAX_VALUE;
498     // if less than 100%, then 10 points for each living cell that fits, -1 point for each cell
          wrong.

500     long hits = 0;
        long misses = 0;
502     long wrong = 0;
        long points = 0;
504     for(int i = 0; i < current.getMap().length; i++){
            for(int j = 0; j < current.getMap().length; j++){
506             boolean curr = current.getMap()[i][j];
                boolean orig = origin.getMap()[i][j];
508             if(curr && orig)
                    hits++;
510             else if(curr && !orig)
                    wrong++;
512             else if(!curr && orig)
                    misses++;
514         }
        }
516     // check if 100%
        if(misses == 0 && wrong == 0){
518         points = Long.MAX_VALUE;
        }
520     else {
            points = hits*10;
522         points -= wrong;
            points -= misses;
524     }
        return points;
526 }
    private final boolean[][] real_map;
528
    /**
530  * Creates a Randomized map of the specified size using the Builtin Random
     * Number Generator. If a seed other than 0 is supplied it will be used for
532  * the RNG.
     * @param size Size of the map, so the map will contain size*size elements.
534  * @param seed Seed for the RNG, 0 = random seed.
     */
536 public Map(int size, int seed)
    {
538     this(size);
        Random rnd = new Random();
540     if(seed != 0)
            rnd.setSeed(seed);
542
        for(int i = 0; i < real_map.length; i++) {
```

```
544             for(int j = 0; j < real_map.length; j++) {
                    if(rnd.nextBoolean())
546                     real_map[i][j] = true;
                }
548         }
            population = this._getPopulation();
550     }

552     public Map(byte[] bytes){
            real_map = fromByte(bytes);
554         population = this._getPopulation();
        }
556
        /**
558      * Creates a Map from the given file, reading it byte for byte.
         * The file MUST be divisible by eight, otherwise the behavior is
560      * undefined.
         * @param filename
562      */
        public Map(File f) throws FileNotFoundException, IOException
564     {
            InputStream is = new FileInputStream(f);
566
            long length = f.length();
568
            if (length > Integer.MAX_VALUE) {
570             throw new IOException("File is too big!: "+f.getName());
            }
572
            if (length*8 % 2 != 0) {
574             throw new IOException("Filesize not divisable by eight!: "+f.getName()+ ":"+length);
            }
576
            // Create the byte array to hold the data
578         byte[] bytes = new byte[(int)length];

580         // Read in the bytes
            int offset = 0;
582         int numRead = 0;
            while (offset < bytes.length
584                && (numRead=is.read(bytes, offset, bytes.length-offset)) >= 0) {
                offset += numRead;
586         }

588         // Ensure all the bytes have been read in
            if (offset < bytes.length) {
590             throw new IOException("Could not completely read file: "+f.getName());
            }
592
            // Close the input stream
594         is.close();

596         real_map = fromByte(bytes);

598     }

600     /**
         * Creates a map using the initial values supplied. Intended to be used for
602      * copying maps.
         * @param initial_values
604      */
        public Map(boolean[][] initial_values)
606     {
            real_map = initial_values.clone();
608         population = this._getPopulation();
        }
610
        /**
612      * Returns a boolean[][] with all the elements of this Map. Can be used to
         * copy maps (See: Map(boolean[][] initial_values).
614      * @return boolean[][] representation of the Map.
         */
616     public boolean[][] getMap(){
            return real_map;
618     }

620     /**
         * Creates an empty Map with the given size. Maps are always square.
622      * @param Size of the map, so the map will contain size*size elements.
         */
624     public Map(int size)
        {
626         real_map = new boolean[size][size];
        }
628
        public boolean getState(int x, int y)
630     {
            if(x >= 0 && x < real_map.length){
632             if(y >= 0 && y < real_map.length){
                    return real_map[x][y];
```

```
634             }
635         }
636         System.err.println("Map.java:getState(" + x + ", " + y + "). L:" + real_map.length);
637         return false;
638     }
639
640     private boolean _getState(int x, int y)
641     {
642         if(x >= 0 && x < real_map.length){
643             if(y >= 0 && y < real_map.length){
644                 return real_map[x][y];
645             }
646         }
647         return false;
648     }
649
650     public boolean _getState(long number)
651     {
652         if(number < 0 || number > numCells()-1){
653             System.err.println("Map.java:_getState(strange number?)");
654             return false;
655         }
656
657         double y = Math.floor(number/(0.0+real_map.length));
658
659         long x = number - (real_map.length*(int)y);
660
661         return _getState((int)y, (int)x);
662
663     }
664
665     public long numCells(){
666         return (real_map.length*real_map[0].length);
667     }
668
669     public boolean setState(int x, int y, boolean state)
670     {
671         if(x >= 0 && x < real_map.length){
672             if(y >= 0 && y < real_map.length){
673                 population += (state?1:-1);
674                 real_map[x][y] = state;
675                 return true;
676             }
677         }
678         System.err.println("Map.java:setState(strange xy?+state).. L:" + real_map.length);
679         return false;
680     }
681
682     public void toggleCell(int x, int y)
683     {
684         if(x >= 0 && x < real_map.length){
685             if(y >= 0 && y < real_map.length){
686                 if(real_map[x][y]){
687                     population--;
688                     real_map[x][y] = false;
689                 }
690                 else {
691                     population++;
692                     real_map[x][y] = true;
693                 }
694                 return;
695             }
696         }
697         System.err.println("Map.java:toggleCell(strange xy?)..");
698     }
699
700     public byte getNeighbours (int x, int y){
701         if(x >= 0 && x < real_map.length){
702             if(y >= 0 && y < real_map.length){
703                 byte count = 0;
704
705
706                 for(int i= x-1; i <= x+1; i++){
707                     for(int j=y-1; j <= y+1; j++){
708                         if(i == x && j == y)
709                             continue; // skip the center
710                         if(_getState(i, j))
711                             count++;
712                     }
713                 }
714                 return count;
715             }
716         }
717         System.err.println("Map.java:getNeighbours("+x+","+y+")..");
718         return 0;
719     }
720
721     public int getPopulation(){
722         return population;
723     }
```

```
724
    public int _getPopulation(){
726     int count = 0;
        for(int i = 0; i < real_map.length; i++){
728         for(int j = 0; j < real_map.length; j++){
                if(real_map[i][j])
730                 count++;
            }
732     }
        return count;
734 }

736 //@SuppressWarnings("unchecked")
    @Override
738 public Iterator<Boolean> iterator() {
        return new MapIterator(this);
740 }
}
```

7.5 ThreadedColourBacktracer

```
1  public Map run(){
       // Initiate..
3
       int numAlmost = 0;
5
       colorMap = new ColorMap(origin.getPopulation(),_ruleB,_ruleS,origin);
7
       if(this.startColors != null){
9          // We want to resume..?
           boolean could = colorMap.tryGoto(startColors);
11
           if(!could)
13         {
               System.err.println("Invalid␣start␣position!");
15             return null;
           }
17         System.out.append("Starting␣from:␣");
           colorMap.print();
19     }

21
       boolean[][] dontCares = null;
23
       // Start rolling..
25     long startTime = System.currentTimeMillis();
       long lastPrint = 0;
27     byte[] last = null;
       long exploration = 0;
29     long exploitation = 0;

31     ColorInstance best = null;

33
       int nextval = 100;
35     boolean second = false;

37     byte size = (byte)origin.getMap().length;

39
       // Agenda, a list of colorMap iterations worth checking out..
41     Agenda agenda = new Agenda();
       AgendaKeeper keeper = new AgendaKeeper(colorMap,agenda);
43
       Thread keeperThread = new Thread(keeper);
45     keeperThread.setPriority(keeperThread.getPriority()+1);
       // since its a slow process, give it its own thread so it can run uninterrupted (hopefuly)...
47     keeperThread.start();

49
       LinkedList<PMSolver> PMJobs = new LinkedList<PMSolver>();
51
       Position[] dots = new Position[origin._getPopulation()];
53     {
           LinkedList<Position> nn = new LinkedList<Position>();
55         PopulationIterator ip = new PopulationIterator(origin);
           while(ip.hasNext()){
57             ip.next();
               int[] xy = ip.getXY();
59             nn.add(new Position(xy[1],xy[0]));
           }
61         for(int i = 0; i < nn.size(); i++)
               dots[i] = nn.get(i);
63     }

65     int numJobs = 50;
```

```
67          while(true){
                if(System.currentTimeMillis()-lastPrint > 4000){
69                  colorMap.print();
                    lastPrint = System.currentTimeMillis();
71                  if(second){
                        System.out.println("Speed: " +
                                keeper.count.doubleValue()/((0.0+System.currentTimeMillis()-startTime)/1000));
73                      System.out.println("Agenda size: "+agenda.size()+"/"+keeper.desiredAgendaSize+ ", 
                                Jobs: "+PMJobs.size()+"/"+numJobs);
                    }
75              }

77

                if(agenda.size() == 0 && PMJobs.size() == 0 && !keeperThread.isAlive()){
79                  // Finished??
                    System.out.println("Done!");
81                  break;
                }
83
                if(PMJobs.size() < 10){
85                  // Create some jobs?

87                  if(PMJobs.size() == 0 && agenda.size() > 1000 && numJobs < 50000){
                        // Increase the ammount of jobs..
89                      numJobs *= 2;
                    }
91
                    for(int i = 0; i < numJobs; i++){
93
                        Byte[] c1 = agenda.get();
95                      PermMap p1 = null;
                        Thread t1 = null;
97
                        if(c1 != null){
99
                            if(dontCares == null)
101                             p1 = new PermMap(c1, dots, size);
                            else
103                             p1 = new PermMap(c1, dots, size, dontCares);

105                         if(staticMap != null){
                                p1.setStaticMap(staticMap);
107                         }

109                         t1 = new Thread(p1);
                            PMJobs.add(new PMSolver(p1,t1));
111                         t1.start();
                        }
113                     else {
                            synchronized(keeper){
115                             if(keeper.desiredAgendaSize < 500000)
                                    keeper.desiredAgendaSize *= 2;
117                             keeper.notify();
                                break;
119                         }
                        }
121                 }
                }
123             else if(agenda.size() == 0 && numJobs > 50){
                    // Reduce the number of jobs
125                 numJobs /= 2;
                }
127
                if(!PMJobs.isEmpty()){
129                 // Any job finished?
                    LinkedList<PMSolver> toRemove = new LinkedList<PMSolver>();
131                 for(PMSolver job : PMJobs){
                        if(!job.t.isAlive()){
133                         byte test = job.pm.runValue;
                            toRemove.add(job);
135                         if(test == -1){
                                // Check if we got a solution here..
137

                                Map solution = job.pm.buildSolution();
139

                                if(solution != null)
141                             {

143                                 Map testMap = Map.nextGeneration(solution,_ruleB,_ruleS);
                                    long diff = Map.diff(testMap, origin, dontCares);
145                                 if(diff == 0){
                                        currentSolution = solution.clone();
147                                     if(fast){
                                            System.out.println("Found a solution @ 
                                                    "+keeper.count.toString());
149                                         return solution;
                                        }
151                                     else {
                                            // print all the solutions..
153                                         System.out.println("Here comes the solution, "+ "cbytes: 
```

```
                                                "+solution.compressMap());
                                        solution.printMap();
155                                     colorMap.print();
                                        System.out.println("E:␣
                                                "+(currentSolution._getPopulation()+0.0f)/(currentSolution.numCells()/2.0f));
157                                 }
                                }
159                             else {
                                    System.out.println("Got␣diff:␣"+diff);
161                                 solution.printMap();
                                }
163                             synchronized(Randum.solutions){
                                    Randum.solutions.add(solution);
165                             }
                            }
167                         numAlmost++;
                        }
169                 }
                }
171             for(PMSolver job : toRemove){
                    PMJobs.remove(job);
173             }
            }
175
        }
177
        System.out.println("All␣colors␣checked,␣Permutations:␣"+keeper.count.toString());
179     return currentSolution;
    }
```

7.6 Reversible Cellular Automata

Only most central methods included.

7.6.1 Second-Order CA

```
    /// For second-order CA, the method to go to the next generation.
2   /// current is the configuration at time t, previous is t-1.
    /// This function calculates t+1. And then sets 't-1 <- t' and 't <- t+1'
4   public void gotoNext(){
        // Rule to use is specified as two byte arrays. Here the rule used is B135/S024
6       Map here = Map.nextGeneration(current, new byte[]{1,3,5}, new byte[]{0,2,4});
        for(int x = 0; x < current.getMap().length; x++){
8           for(int y = 0; y < current.getMap().length; y++){
                boolean curr = here.getState(x, y);
10              boolean prev = previous.getState(x,y);
                if(prev != curr){
12                  next.setState(x, y, true);
                }
14
            }
16      }

18      previous = new Map(current.clone().getMap());
        current = new Map(next.clone().getMap());
20  }
```

7.6.2 Block Cellular Automata

```

2   /// For block Cellular Automata, the method to go to the next generation.
    /// the parameters are; m, the current configuration
4   ///                     odd, if the current generation is divisable by 2
    ///                     reverse, if the local block rule should be applied in reverse.
6   /// Note that odd controls the partitioning of the Block CA so that it will alternate as it should.
    ///     reverse must be used to evolve the CA backwards in time as the local rule must be reversed.
8   public static Map next(Map m, boolean odd, boolean reverse){
        int length = m.getMap().length;
10      Map tmp = new Map(length);

12      int n = length/2;

14      int row = 0;
        int column = 0;
16
        boolean[] here = new boolean[4];
18      int nn = n*n;

20      int r2 = 0;
        int c2 = 0;
```

```
22          for(int square = 0; square < nn; square ++){
                //iterates over bca squares..
24
                if(odd){
26                  row = (int)Math.floor((0.0+square)/n);
                    column = square - (n*row);
28                  r2 = row*2;
                    c2 = column*2;
30
                    here[0] = m.getState(wrapMinus(c2,3), wrapMinus(r2,3)); // wrapMinus is a simple helper
                        function defined below.
32                  here[1] = m.getState(wrapMinus(c2,3), r2);
                    here[2] = m.getState(c2, wrapMinus(r2,3));
34                  here[3] = m.getState(c2, r2);

36                  here = critterBlock(here, reverse); // By changing the local rule the Block Cellular
                        Rule will change
                    //here = tronBlock(here); // Both the Critters and Tron local rules are defined below.
38
                    tmp.setState(wrapMinus(c2,3), wrapMinus(r2,3), here[0]);
40                  tmp.setState(wrapMinus(c2,3), r2, here[1]);
                    tmp.setState(c2, wrapMinus(r2,3), here[2]);
42                  tmp.setState(c2, r2, here[3]);
                }
44              else { // even
                    row = (int)Math.floor((0.0+square)/n);
46                  column = square - (n*row);
                    r2 = row*2;
48                  c2 = column*2;

50                  here[0] = m.getState(c2, r2);
                    here[1] = m.getState(c2, r2+1);
52                  here[2] = m.getState(c2+1, r2);
                    here[3] = m.getState(c2+1, r2+1);
54
                    here = critterBlock(here, reverse);
56                  //here = tronBlock(here);

58                  tmp.setState(c2, r2, here[0]);
                    tmp.setState(c2, r2+1, here[1]);
60                  tmp.setState(c2+1, r2, here[2]);
                    tmp.setState(c2+1, r2+1, here[3]);
62              }
            }
64          return tmp;
        }
66
        /// Helper function, basicaly a decrement operator that wraps to the given maximum.
68      private static int wrapMinus(int cur, int max){
            if(cur - 1 < 0) return max;
70          return cur-1;
        }
```

7.6.3 Tron Local Rule

```
2       /// Tron local rule. Given the binary states of the 4 cells in a block,
        /// the Tron local rule is applied and the states returned.
4       /// The Tron rule simply inverts all cells if all of them have the same state,
        /// otherwise no change is made.
6       public boolean[] tronBlock(boolean[] input){
            byte countOn = 0;
8           for(byte i = 0; i < input.length; i++){
                if(input[i])
10                  countOn++;
            }
12          if(countOn == 0){ // All cells dead -> inverted -> all cells live
                return new boolean[]{true, true, true, true};
14          }
            else if(countOn == 4){ // All cells live -> inverted -> all cells dead
16              return new boolean[]{false, false, false, false};
            }
18          else // no change
                return input;
20      }
```

7.6.4 Critters Local Rule

```
1
        /// Critters local rule. Given the binary states of the 4 cells in a block,
3       /// the Critters local rule is applied and the states returned.
        /// The Critters rule inverts all cells unless there are exactly 2 live cells.
5       /// If there are 3 live cells the block is also rotated 180 degrees (turned up-side down).
        /// Note that in reverse the rotation is aplied to blocks with 1 live cell instead of 3.
7       public static boolean[] critterBlock(boolean[] input, boolean reverse) {
            byte numAlive = 0;
9           int rotaOne = 0; // for calculating the rotation
```

```
        for (int i = 0; i < input.length; i++) {
            if (input[i]) {
                numAlive++;
                if(reverse)
                    rotaOne = i;
            } else if (!reverse) {
                rotaOne = i;
            }
        }

        if (numAlive == 2) {
            return input;
        }

        if ((!reverse && numAlive == 3) || (reverse && numAlive == 1)) {
            //rotate
            boolean tmp = false;
            switch (rotaOne) { // rotation, based on the position of the rotaOne cell
                case 0:
                case 3:
                    tmp = input[0];
                    input[0] = input[3];
                    input[3] = tmp;
                    break;
                case 1:
                case 2:
                    tmp = input[1];
                    input[1] = input[2];
                    input[2] = tmp;
                    break;
            }
        }

        //invert
        for (int i = 0; i < input.length; i++) {
            if (input[i]) {
                input[i] = false;
            } else {
                input[i] = true;
            }
        }
        return input;
    }
```

7.7 GACABacktracer

Only most central methods included.

7.7.1 Tournament Selection

```
    public int tournamentSelection()
    {
        int numToCompete = 3;
        // only 1 victor..
        int best = -1;
        long score = Long.MIN_VALUE;
        for(int i = 0; i < numToCompete; i++){
            int current = (int)(Math.random()*population_size); // Selects a random gene to compete
            if(fitness[current]>score){
                best = current;
                score = fitness[current];
            }
        }
        return best;
    }
```

7.7.2 Random Crossover

```
    public Map randomCrossover(Map map1, Map map2){
        // Creates offspring taking randomly from parent1 or parent2 (map1 or map2).

        Map offspring = new Map(map1.getMap().length);

        MapIterator oit = (MapIterator)offspring.iterator();

        MapIterator iter1 = (MapIterator)map1.iterator();
        MapIterator iter2 = (MapIterator)map2.iterator();

        while(oit.hasNext()){
            oit.next();
            boolean p1 = (Boolean)iter1.next();
            boolean p2 = (Boolean)iter2.next();
```

```
15
            if(((int)(Math.random()*2)) == 0){
17              if(p1) oit.toggle();
            }
19          else {
                if(p2) oit.toggle();
21          }
        }
23      return offspring;
    }
```

7.7.3 Edge Flip Mutation

```
    public Map edgeFlipMutate(Map map) {
2       // Flip mutate only bits with neighbours
        MapIterator it = (MapIterator) map.iterator();
4       int x,y;
        while(it.hasNext()){
6           it._next();
            x,y = it.getXY();
8
            if(map.getNeighbours(x, y) > 0 && Math.random() <= mutation_chance)
10              map.toggleCell(x, y);
        }
12      return map;
    }
```